Towards a Formal Methods Body of Knowledge for Railway Control and Safety Systems

FM-RAIL-BOK Workshop 2013

Gruner, Stefan; Haxthausen, Anne Elisabeth; Maibaum, Tom ; Roggenbach, Markus

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Technical University of Denmark

Towards a Formal Methods Body of Knowledge for Railway Control and Safety Systems

FM-RAIL-BOK Workshop 2013
Madrid, Spain, September 2013
Proceedings

Stefan Gruner,
Anne E. Haxthausen,
Tom Maibaum,
Markus Roggenbach (Eds.)

DTU Compute Technical Report-2013-20

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

ISSN: 1601-2321
ISBN: 978-87-643-1303-1

Contents

Towards a formal methods body of knowledge for railway control and safety systems

Stefan Gruner
University of Pretoria, South Africa

Tom Maibaum
McMaster University, Canada

Anne Haxthausen
Technical University of Denmark

Markus Roggenbach
Swansea University, Wales, UK

Formal methods in software science and software engineering have existed at least as long as the term "software engineering" (NATO Science Conference, Garmisch, 1968) itself. Its various methods and techniques include algebraic specification, process-algebraic modelling and verification, Petri nets, fuzzy logics, etc. Especially in railway control and safety systems, formal methods have reached a considerable level of maturity. For example, the B-method has been used successfully to verify the most relevant parts of the Metro underground railway system of the city of Paris (France). Thus, it appears timely to begin the compilation of a so-called body of knowledge (BoK) dedicated to this specific area.

The FM-RAIL-BOK WORKSHOP 2013 (see also http://ssfmgroup.wordpress.com), held on 23 September 2013 in Madrid, was a first successful step towards this aim. This international workshop was affiliated to the SEFM 2013, the 11th International Conference on Software Engineering and Formal Methods, Madrid. This volume compiles two abstracts and seven contributed papers of talks presented at the workshop.

As keynote, Alessandro Fantechi, Universita di Firenze, presents an overview "Twenty-Five Years of Formal Methods and Railways: What Next?". This keynote was shared with the 18th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'13). Towards our aim of compiling a body of knowledge, Tom Maibaum, McMaster University, reflects upon "BoKs and Engineering Knowledge".

All contributed papers were reviewed by the workshop PC. They cover topics as varied as Ontologies, Verification of Data and Designs for Railway Control Systems, Verification of Interlocking Programs Expressed in Ladder Logic, and ECTS/ERTMS Modelling. Not necessarily presenting new scientific results, these papers compile case-based "best practice" knowledge in the spirit of classical engineering handbooks. A selection of these papers in improved and extended versions will appear in a volume of the Springer LNCS series.

As FM-RAIL-BOK co-chairs we would like to thank all authors who submitted their papers to our workshop, Alessandro Fantechi for accepting our invitation to present a keynote, the workshop participants, our Programme Committee, Manuel Nunez, Universidad Complutense de Madrid, Spain, and Steve Counsel, Brunel University, United Kingdom, for the smooth cooperation with SEFM'13, Linh Vu Hong for valuable help in preparing these proceedings, and Erwin R. Catesbeiana (Jr) for help with workshop organization on the fly.

The FM-RAIL-BOK co-chairs
November 2013

Twenty-five years of formal methods and railways: what next?

(Invited Paper)

Joint FMICS/FM-RAIL-BOK Keynote Speaker.

Reprinted from *C. Pecheur and M. Dierkes (eds.), Formal Methods for Industrial Critical Systems, LNCS 8187, page ix, Springer-Verlag, 2013,* with permission of Springer-Verlag.

Alessandro Fantechi
DINFO - University of Florence
Via S. Marta 3
Firenze, Italy
Email: fantechi@dsi.unifi.it

Abstract

Railway signaling is now since more than 25 years the subject of successful industrial application of formal methods in the development and verification of its computerized equipment.

However the evolution of the technology of railways signaling systems in this long term has had a strong influence on the way formal methods can be applied in their design and implementation. At the same time important advances had been also achieved in the formal methods area.

The evolution of railways signaling systems has seen railways moving from a protected market based on national railway companies and national manufacturers to an open market based on international standards for interoperability, in which systems of systems are providing more and more complex automated operation, but maintaining, and even improving, demanding safety standards.

The scope of the formal methods discipline has enlarged from the methodological provably correct software construction of the beginnings to the analysis and modelling of increasingly complex systems, always on the edge of the ever improving capacity of the analysis tools, thanks to the technological advances in formal verification of both qualitative and quantitative properties of such complex systems.

In spite of these advances, the verification of complex railway signalling systems is still a main challenge and an important percentage of the cost in the development of these systems. We will discuss a few examples of such systems that witness these difficulties.

The thesis we will put forward in this talk is that the complexity of future railway systems of systems can be addressed with advantage only by a higher degree of distribution of functions on local interoperable computers - communicating by means of standard protocols - and by adopting a multi-level formal modelling suitable to support the verification at design time and at different abstraction levels of the safe interaction among the distributed functions.

What IS a BoK?

– extended abstract –

Tom Maibaum
McMaster Centre for Software Certification
McMaster University 1280 Main St W, Hamilton, ON, Canada L8S 4K1
Email: tom@maibaum.org

I. Main points

Software engineering is different from traditional engineering disciplines in certain crucial ways. But software engineering *is* an engineering discipline. However, software engineering fails to meet the requirements of an engineering discipline, as commonly conceived by conventional engineers. Software Engineering Books of Knowledge (BoKs) fail spectacularly in organising engineering knowledge as understood in classical engineering disciplines.

"The SWEBOK Guide:

- characterizes the contents of the software engineering discipline
- promotes a consistent view of software engineering worldwide
- clarifies software engineering's place with respect to other disciplines
- provides a foundation for training materials and curriculum development, and
- provides a basis for certification and licensing of software engineers."

We will "show" below that this is nothing like classical engineering knowledge and, in particular, like the so called cookbooks well known in engineering.

II. What is engineering?

So, what characterises classical engineering disciplines? The following books have been immensely helpful in understanding engineering:

- GFC Rogers, The Nature of Engineering, The Macmillan Press Ltd, 1983
- WG Vincenti, What Engineers Know and How They Know It, The Johns Hopkins University Press, 1990

We have also been inspired by various papers of Michael Jackson. [1] That software engineering is an engineering discipline is a simple consequence of the fact that: "engineering refers to the practice of organising the design and construction of any artifice which transforms the physical world around us to meet some recognised need" [2]. Vincenti [3] argues that engineering is different, in epistemological terms and, consequently, as praxis, from science or even applied science: "In this view, technology, though it may apply science, is not the same as or entirely applied science". Rogers argues the same view on the basis of what he calls the teleological distinction concerning the aims of science and technology: "In its effort to explain phenomena, a scientific investigation can wonder at will as unforeseen results ... The essence of technological investigations is that they are directed towards serving the process of designing and constructing particular things whose purpose has been clearly defined." "We have seen that in one sense science progresses by virtue of discovering circumstances in which a hitherto acceptable hypothesis is falsified, and that scientists actively pursue this situation. Because of the catastrophic consequences of engineering failures - whether it be human catastrophy [sic] for the customer or economic catastrophy [sic] for the firm - engineers and technologists must try to avoid falsification of their theories. Their aim is to undertake sufficient research on a laboratory scale to extend the theories so that they cover the foreseeable changes in the variables called for by a new conception. The scientist seeks revolutionary change - for which he may receive a Nobel Prize. The engineer too seeks revolutionary conceptions by which he can make his name, but he knows his ideas will not be taken up unless they can be realised using a level of technology not far removed from the existing level." [2]

So, science is different from engineering. We can ask what the praxis of engineering is. Vincenti [3] defines engineering activities in terms of *design*, *production* and *operation* of artefacts. Of these, design and operation are highly pertinent to software engineering. In the context of discussing the focus of engineers activities, he then talks about *normal design* as comprising "the improvement of the accepted tradition or its application under new or more stringent conditions". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has good likelihood of accomplishing the desired task." Jackson discusses this concept of normal design, although he does not use this phrase himself: "An engineering handbook is not a compendium of fundamental principles; but it does contain a corpus of rules and procedures by which it has been found that these principles can be most easily and effectively applied to the particular design tasks established in the field. The outline design is already given, determined by the established needs and products." "In this context, design innovation is exceptional. Only once in a thousand car designs does the designer depart from the accepted structures by an innovation like front-wheel drive or a transversely positioned engine. True, when a radical innovation proves successful it

becomes a standard design choice for later engineers. But these design choices are then made at a higher level than that of the working engineer: the product characteristics they imply soon become well understood, and their selection becomes as much a matter of marketing as of design technology. Unsuccessful innovations - like the rotary internal combustion engine - never become established as possible design choices." "The methods of value are micro-methods, closely tailored to the tasks of developing particular well-understood parts of particular well-understood products."

Another important aspect of engineering design is the organising principle of hierarchical design: "Design, apart from being normal or radical, is also multilevel and hierarchical. Interesting levels of design exist, depending on the nature of the immediate design task, the identity of some component of the device, or the engineering discipline required." [3] It is quite clear from the engineering literature that engineering normally involves the use of multiple technologies. The observation that software engineering requires knowledge of other domains and that its teaching should be application oriented is not as perspicacious as its proponents would have us believe. This is part of the essence of engineering, whatever the discipline. An implied but not explicitly stated view of engineering design is that engineers normally design devices as opposed to systems, in the sense of Vincenti. A device, in this sense, is an entity whose design principles are well defined, well structured and subject to normal design principles. (See also Michael Polanyis operational principle of a device [4].) A system, in this sense, is an entity that lacks some important characteristics making normal design possible. "Systems are assemblies of devices brought together for a collective purpose." Examples of the former given by Vincenti are airplanes, electric generators, turret lathes; examples of the latter are airlines, electric-power systems and automobile factories. The software engineering equivalent of devices may include compilers, relational databases, PABXs, etc. Software engineering examples of systems may include air traffic control systems, automotive software, the internet, etc. It would appear that systems become devices when their design attains the status of being normal. That is, the level of creativity required in their design becomes one of systematic choice, based on well defined analysis, in the context of standard definitions and criteria developed and agreed by engineers.

III. Engineering knowledge

Is the knowledge used by software engineers different in character from that used by conventional engineers? The latter is underpinned by mathematics and some physical science(s), providing models of the physical universe in terms of which artefacts must be understood. What about software engineering? I would claim that logic (in its widest sense) fulfills these roles, although from different perspectives in computer science and software engineering. Software engineering is distinguished from conventional engineering because the artefacts constructed by the former are conceptual, while those built by the latter are physical. For the latter, the "real world" is a fixed constraint, whereas it is not clear that there are the same limitations on the "computational world". There is an existing track record of working with concepts and abstractions in mathematics and logic, particularly philosophical logic.

What distinguishes software engineering is the day to day invention of theories (descriptions) by engineers and the problems of size and structure induced by the nature of the artefacts. Can we successfully apply the analogy between conventional engineering and its use of mathematical techniques and scientific analyses, on the one hand, and software engineering and its use of ideas from the relevant mathematics and logic based analyses, on the other?

An example that may be used in this context is program construction. The well understood underlying mathematics was developed over 25 years (in the sequential case), starting in the 1960s. Thus, we might have expected the SE equivalent of the engineering CAD tool to appear at the end of this time. Instead, we have CASE tools with no relation to the underlying mathematics, or formal methods, which offer a relaxation of the exhaustiveness requirement of the scientific/theoretical viewpoint. There is no equivalent of the conventional engineering disciplines available in industrial software engineering settings.

IV. Categories of engineering knowledge

Software engineering is distinct in character from conventional disciplines of engineering. However, it has enough in common with them to look for the same categories of knowledge [3]:

1) Fundamental design concepts
2) Criteria and specifications
3) Theoretical tools
4) Quantitative data
5) Practical considerations
6) Design instrumentalities

Fundamental design concepts include the operational principle of their device. According to Polanyi, this means knowing for a device "how its characteristic parts ... fulfill their special functions in combining to an overall operation which achieves the purpose". [4] A second principle taken for granted is the normal configuration for the device, i.e., the commonly accepted arrangement of the constituent parts of the device. These two principles (and possibly others) provide a framework within which normal design takes place. Criteria and specifications allow the engineer using a device with a given operational principle and normal configuration to "translate general, qualitative goals couched in[to] concrete technical terms". That the development of such criteria may be problematic is clear. However, the development and acceptance of such criteria is an inherent part of the development of engineering disciplines.

Engineers require theoretical tools to underpin their work, including intellectual concepts for thinking about design, as well as mathematical methods and theories for making design calculations. Both conceptual tools and mathematical tools may be devised specifically for use by the engineer and be of no particular value to a scientist/mathematician. "... the most useful context for the precision and reliability that formality can offer is in sharply focused micro-methods, supporting specialised small-scale tasks of analysis and detailed design." [1] Engineers also use quantitative data as well as tabulations

of functions in mathematical models. (A good example in software engineering of this thoroughness in providing data useful for design is the work of Knuth on sorting and searching.)

There are also practical considerations in engineering. These are not usually subject to systematisation in the sense of the categories above, but reflect pragmatic concerns. For example, a designer will use various trade-offs which are the result of general knowledge about the device, its use, its context, its cost, etc. Design instrumentalities include "the procedures, ways of thinking, and judgmental skills by which it [design] is done" [3]. This is clearly what the Capability Maturity model has in mind when it refers to well defined and repeatable processes in software engineering.

According to Vincenti, as noted above, the day to day activities of engineers consist of normal design, as comprising "the improvement of the accepted tradition or its application under new or more stringent conditions". This is the combination of discipline and a little bit of creativity encapsulated in engineering cookbooks! He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

V. In summary

"An engineering handbook is not a compendium of fundamental principles; but it does contain a corpus of rules and procedures by which it has been found that these principles can be most easily and effectively applied to the particular design tasks established in the field. The outline design is already given, determined by the established needs and products." [1]

Systems become devices when their design attains the status of being normal, i.e., the level of creativity required in their design becomes one of systematic choice, based on well defined analyses, in the context of standard definitions and criteria developed and agreed by the relevant engineers ([3], definition of normal design). This is exactly what engineering BoKs should be about!

References

[1] M. Jackson, "The operational principle and problem frames," in *Reflections on the Work of CAR Hoare*. Springer, 2010, pp. 143–165.

[2] G. F. C. Rogers, *The nature of engineering: a philosophy of technology*. Macmillan Press, 1983.

[3] W. G. Vincenti, *What engineers know and how they know it: Analytical studies from aeronautical history*. The Johns Hopkins University Press, 1990.

[4] M. Polanyi, *Personal Knowledge: Towards a Post-critical Philosophy.* Routledge & Kegan Paul, London, 1958, reprinted by University of Chicago Press (1974).

An ontology for complex railway systems, application to the ERTMS/ETCS system

Olimpia Hoinaru[†], Georges Mariano[†], Christophe Gransart[‡]
[*] Université Lille Nord de France
French Institute of Science and Technology for Transport,
Development and Networks (IFSTTAR)
[olimpia.hoinaru|georges.mariano|christophe.gransart]@ifsttar.fr
[†] ESTAS
Évaluation des Systèmes de Transports Automatisés et de leur Sécurité
[‡] LEOST
Laboratoire Électronique Ondes et Signaux pour les Transports

***Abstract*—We present hereafter our experimental work of building an ontology of the European Rail Traffic Management System (ERTMS) domain. ERTMS is a railway complex control system defined on the basis of publicly available specification documents, the System Requirements Specification (SRS). We will describe the methodology that we used to define an initial structure for an ERTMS ontology. The main goal of this work is to supply a first formalization of the ERTMS knowledge in order to provide the basis of a later development process i.e validating the specifications, developing the software/hardware components and finally validating the system.**

***Keywords*—Ontologies, ERTMS/ETCS, railway systems**

I. Introduction

ERTMS stands for the European Rail Traffic Management System. This is a European standard for the process control system and signalling and new lines for the replacement of existing systems for conventional lines. ERTMS contains two basic elements:

- GSM-R (Global System for Mobiles - Railway): the communication component containing a voice communication network between vehicle drivers and line controllers. It provides routing and portability for ETCS data. It is based on the GSM public standard with specific features for railways.
- ETCS (European Train Control System): the signalling system component that includes control movement authorities, automatic train protection and interface with the interlocking.

Developing such a complex structure is, of course, a real challenge. Only by considering the development of the corresponding software, we can observe on the figure 1 the general evolution of the technologies employed.

Roughly speaking, in the past, the challenge was to define a method to derive machine code from documentation (this documentation coming from the informal, and sometimes implicite "knowledge" of the system to be developed). To answer this challenge, (countless) modelling methods were defined and are now available. Thus we may now assert that the code is correct because it corresponds to previously established models (whether they are formal or not). Our problematics is how we can provide good models, preferably formal ones.

By doing this, we completely follow the paradigm stated in [1]: "Before software can be designed we must understand the requirements. Before requirements can be finalised we must have understood the domain". But where Dines Bjorner uses pure formal logic to tackle generic sample problems, we will experiment the use of ontological technologies (conceptualization, formalization, reasoning) to tackle a real and complex system.

II. General goal(s)

The work presented in this article is situated at the intersection of several domains i.e. knowledge management and Web semantics, knowledge representation and formalization, as well as system modelling. The knowledge of the ERTMS domain is considered and formalized for understanding and reuse issues.

Several methods (models) can be used to capture the different aspects of a railway complex system. Based on the fact that the same concept can have different meanings in different domains, the need for specification of these semantic differences was felt.

The ERTMS ontology aims at modelling and formalizing the System Requirements Specification documents of the ERTMS. These documents are written in natural language. The aim of this ontology is the formalization of these specifications in order to obtain a data structure that can be reusable in the framework of other research in the ERTMS field. A module of this ontology is the OSI (Open Systems Interconnection) [2] model and another one concerns the application of the OSI model to the ERTMS/ETCS subsystem dealing with the data transmission by means of radiocommunication.

III. Elaborating ontologies

Ontologies are formal representations of knowledge of a certain domain. Several definitions of the term "ontology" have been provided. [3] poses that "an ontology is an explicit specification of a conceptualization". According to the same author "the term is borrowed from philosophy, where an ontology is a systematic account of Existence".

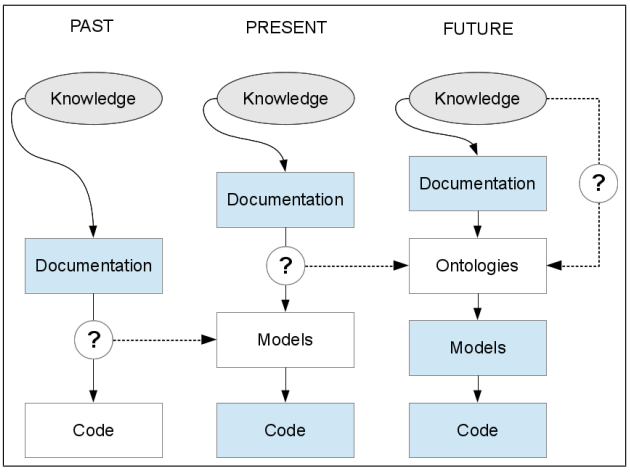


Fig. 1. Ontologies for software-based systems

There are four types of information allowing us to precise what is that we represent in an ontology. These are the type of ontology (domain ontologies, generic ontology, ontology of a method of solving a problem, application ontology and representation ontology), the properties, the "is-a" relation and the other relations [4].

The knowledge of a domain is formalized using several notations with the aim of regrouping and creating a formal structure of the concepts of this domain into a web of knowledge.

We chose an ontology creation tool using the Web Ontology Language (OWL), i.e. the Protégé tool. Protégé-2000 was developed by Mark Musen's group at Stanford Medical Informatics. In this environment, concepts are formalized as classes together with their several types of properties and the relations among them. The so-called "rules" are created for the purpose of modelling requirements and certain "behaviors" of the system.

In the railway domain, documents describing the System Requirements Specifications were issued with the specific aim of explaining and clarifying the usage of a part of the terms/concepts used in this domain, and of the system itself.

A. Approaches

This paragraph presents some of the ontology development methodologies existing. "Methontology" is the term used to describe one of these methodologies for creating an ontology. It is among the more comprehensive ontology engineering methodologies as it is one for building ontologies either from scratch, reusing other ontologies as they are, or by a process of re-engineering them.

But methontology is not the only methodology of creating ontologies. Other methodologies like, for example, the corpus-based methodology exist. In this case, the ontology is derived from documents provided in natural language that can also contain diagrams, flow charts, or tables. It is the case of the ERTMS ontology whose creation we are presenting in this study.

- [5] is a publication dealing precisely with this subject-matter. The authors describe here the reasons that can lead one to develop an ontology i.e. the usage of this kind of structure, its definition, several types of methodologies, as well as the composition and structure of an ontology. We found this article particularly interesting for its explicitness and pedagogical style. The example taken is a test ontology created by the Protege developers, a wine ontology.

IV. Global view of the proposed ERTMS ontology

A. The chosen method

Our ontology is based on normative documentation, i.e. the System Requirements Specification [6] documents provided by the European Railway Agency (ERA). Other related documents are the "ERTMS Glossary" and the "ETCS Implementation Handbook" published by International Union of Railways (UIC). This is an ontology created as a semantic model and module extracted from the below mentioned documents. The extraction is based on the study, the comprehension of these documents, and on the transposition of the information conceptualized in the same documents. All this is being carried out manually by (some of) the authors of this article and not performed automatically as some software can do. As the study of these SRS within the framework of this research is at its beginnings, we chose to start it manually for a better usage of the comprehension of the human understanding. A perspective of this study is the automation of the information extraction from the SRS and other documents. This ontology is a way of formalizing the information provided by these documents. It is not the ultimate aim of this study, but just another more explicit form of the SRS documents.

The railway domain is an environment where numerous heterogeneous information sources exist. The ERTMS system basically relies on information exchange. Ontologies provide a number of useful features for intelligent systems, as well as for knowledge representation generally. The ERTMS ontology that we propose also aims at offering a solution for information exchange, and this for a better railway transportation world.

Train control is an important part of any railway operation management system. In the past a number of different Automatic Train Control (ATC) systems have evolved in different countries at different times. Due to the incompatibility and lack of interoperability among these systems, as well as to a significant increase in density of train traffic anticipated, many railways rethink their infrastructure strategy, in order to accommodate high levels of traffic, in which ATC systems play an important part. This and the fact that many railway systems would like to introduce standardized components to reduce system costs are, among others, the reasons of the existence of this system. In order to establish international standardization of ATC systems, the SRS document specifies the European Rail Traffic Management System/European Train Control System (ERTMS/ETCS).

The ERTMS System Requirements Specification is a set of documents written in natural language, English in this case. It specifies the European Rail Traffic Management System/ European Train Control System (ERTMS/ETCS) which is a control and signalisation innovative system of the railway

vehicles and tracks. Also, system safety plays an important role in railway transport as it constitutes a challenging issue that has engaged strong and continuous research interest.

B. Ontology building from normative documentation

As mentioned before, in this ERTMS ontology, concepts are formalized as classes (terms). An ontology is not only the identification and classification of concepts, but also of their inherent characteristics that are here called "properties". Moreover the relations gather the concepts together. Primarily, we used the "is-a" relation which is a subsumption relation allowing the formal heritage of properties. The "has-a" relation, also known as composition, is used as well in this ontology, this time not for the class layer but for the instance layer. If, at the beginning, we had conceived our primary concept structure using the two relations for the classes, a differentiation became crucial as work proceeded. Then, other relations were established according to the system's syntax. These relations are created based on properties declaration and domain specification (tab allowing to select the class(es)) on which they take effect. Our ontology is structured into several modules.

- the `Entity` module, i.e. the superclass containing several entities like `Driver`, `ERTMS`, `Procedure`, describes entities that are used to define the required system behavior on a context level.
- the OSI_Model is a sibling class of `Entity`, a module aiming at describing the Open Systems Interconnection (OSI) model. This is a conceptual model that characterizes and standardizes the internal functions of a communication system by partitioning it into abstraction layers. This module will be more thoroughly explained in section VI.
- another sibling class of the above mentioned one is `Source`. It formalizes information about the SRS and other ERTMS/ETCS documents used as corpus of these ontology.
- `TrainCategories` is also a child of the `Entity` superclass, containing information about the different types of rolling stock.

V. Modelling ERTMS procedures

In figure 2, we present an example of a procedure defined in the SRS called "Entering SH mode". The "Shunting" mode is, by definition, a type of ERTMS/ETCS on-board equipment allowing a train to move without having the update train data.

There exist several ERTMS operating modes, as well as all operational modes and procedures necessary to ensure safe information exchange between the driver and the embedded subsystem. Each mode is associated with a specific configuration (train, track and conditions) defining the system state.

Transitions between modes require the establishment of different conditions required to perform the transition properly, i.e. safely. In the SRS, the procedures associated or involving mode transistions are defined by flowcharts linking conditions, decisions and states. The "shunting" flowchart is presented in figure 2.

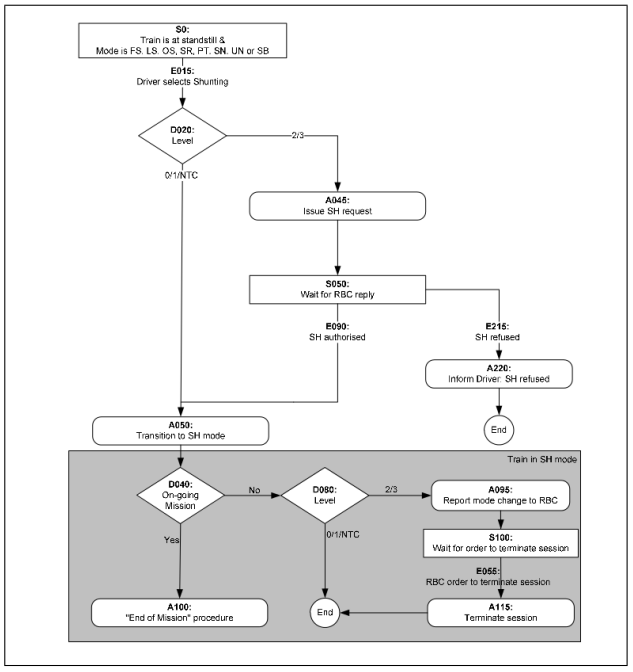


Fig. 2. Flowchart for the "Shunting" procedure

In order to catch the semantics of these flowcharts in our ontology, we transformed the state transitions in each flowcharts into rules expressed in the SWRL language provided by the Protégé framework. SWRL stands for "Semantic Web Rule Language" and provides a syntax and a semantics to express rules upon the entities available in the ontology. SWRL rules have the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold.

Considering the flowcharts as state-transition machines, we derived a flowchart into a set of SWRL rules, each rule corresponding to a transition. By doing this, we intended to catch the dynamic behavior of ERTMS/ETCS control system.

VI. Focus on the radio-communication part

As mentioned in the sections before, this ontology is constructed by modules. One of these modules formalizes the OSI (Open Systems Interconnection) model and another sub-module deals with the application of the generic OSI model to the ERTMS system. This section presents the generic telecommunication model, followed by its instanciation with the OSI model and finally with the ERTMS telecommunication subsystem.

A. The radio telecommunication model

First, we defined a generic radio telecommunication model. This model/module is composed of several concepts[1]:

[1]concepts defined in our ontology will be typesetted like this `ConceptName`

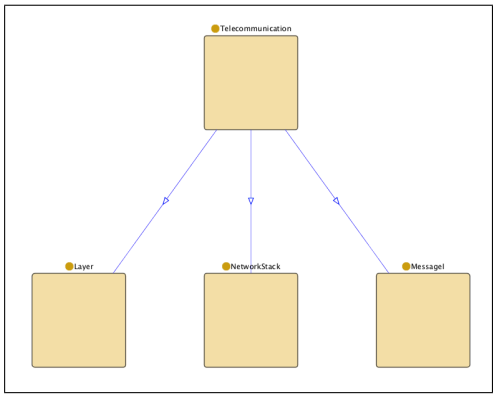


Fig. 3. Generic radio telecommunication concepts

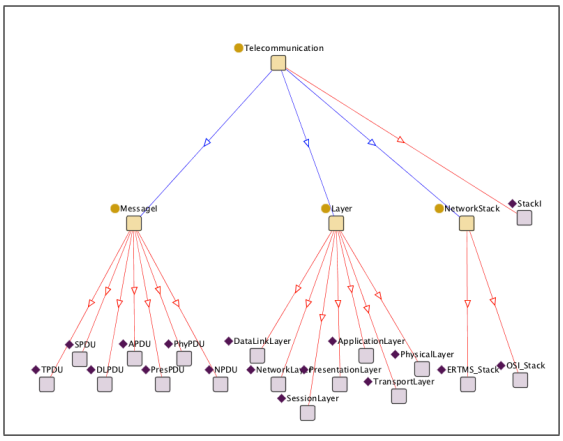


Fig. 4. Hierarchy and instances for the OSI model

- the `NetworkStack` is the telecommunication stack which is composed of several `Layers`.
- the `Layer` is a part of the `NetworkStack` which is able to marshall and unmarshall some Messages. Each `layer` is linked to two other `Layers`: an upper layer and a down layer. The combination of this set of layers is a `NetworkStack`. A `layer` manipulates some `Messages`.
- the `Message` defines the data that will be sent and received on the network by the `Layers`.
- The `Telecommunication` concept references the concepts defined above.

Figure 3 presents graphically this set of concepts.

B. Feeding the ontology with the OSI model

Next, we populated the ontology with the concepts that describe the OSI model composed of 7 layers. This part of the work was usefull to see if the concepts defined into the radio telecommunication model were enough and to be sure that nothing was forgotten.

Figure 4 presents the hierarchy as defined previously and all the instances which represent the different layers of a classical OSI network stack. There is one important relation between

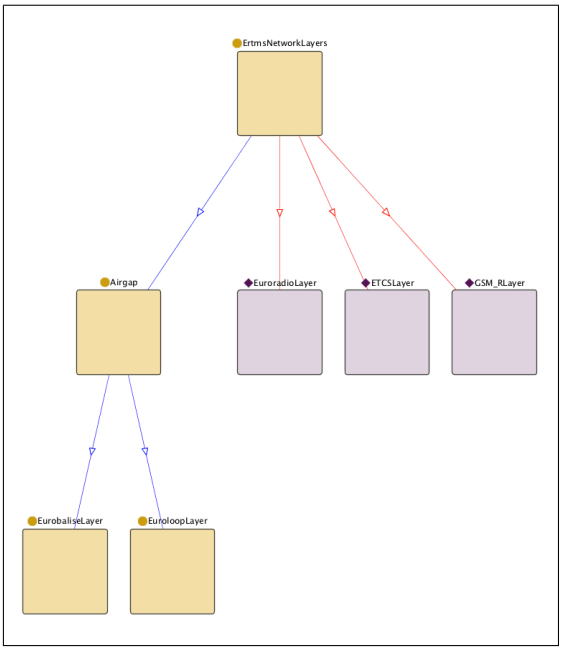


Fig. 5. Hierarchy and instances for the ERTMS radio subsystems

the layers. That relation describes the link between two consecutive layers. Its is notated `hasUpperLayer` between layer N and N+1 and its opposite `hasDownLayer` between N and N-1. The uppest layer does not have an `hasUpperLayer`, nor does the lowest layer have a `hasDownLayer`. These relations are not shown on the figure 4 to keep a clear schema.

C. Feeding the ontology with the ERTMS radio subsystem

We applied the same reasoning to represent the concepts of the ERTMS/ETCS radio subsystem. This radio subsystem is composed of three layers (from down layer to upper layer):

- the `GSM_RLayer` is based on the GSM specification with some modifications to fit the railway industry needs. The goal of this layer is to transport data packets through a celullar network between the train and the Radio Block Center (RBC).
- the `EuroradioLayer` deals with the end to end communication between an embedded application into the train and an application on ground. This layer is also responsible for non functional properties like authentication and crytptography of the messages.
- the `ETCSLayer` manages the messages at the application level of ETCS. This layer permits the communication between the onboard EVC and the ground system RBC that gives the movement authority grant(s) to the train.
- `AirGap`, `EurobaliseLayer` and `EuroloopLayer` represent equipements put on the track. These equipements communicate with the train when it goes over the equipments.

Figure 5 shows the ERTMS Network layer stack with three instances that correspond to the layers described just before.

D. Current state

This ERTMS ontology is structured into several layers. The `Thing` superclass contains several classes like `Entity`, `Source`, `OSIModel`, etc which, in their turn, have several subclasses. For example, the `Entity` class reunites the subclasses `Driver`, `ERTMS` and `Procedure`. The `ERTMS` subsubclass contains `ApplicationLevel`, `ERTMSNetworkLayer` and `ETCS`. These are just a few examples of terms that we enterred in the surface levels of the class structure of this ontology.

Currently, the ERTMS ontology that we have been creating contains 112 classes, 193 instances, and 104 properties including object, datatype and annotation properties.

VII. Related works

Due to the lack of space, we won't provide a huge panel of related works. With a few references, we will show that the main aspects of our work have already been studied and that there exist a solid background to tackle now with complex railway systems (like ERTMS/ETCS is) while involving several concerns like formalisation, requirements engineering, traceability, ...

A. Ontologies and software engineering

- In [7], an ontology called *OntoTest* is presented. This ontology is developed in order to promote organization, reuse and sharing of software testing knowledge. The main concepts and artefacts of testing are described (Process, phases, resources, procedures). The ontology itself is figured with UML class diagrams, W3C formalisms are not used in the paper but the ontology is now available in OWL format.
- The work presented in [8] is very close to the goals of our work. Starting from an industrial-use case (the Onboard Unit of ERTMS) a methodology to improve the testing process is provided. This methodology involves the analysis of the SRS specifications, the rewriting of the requirements into a "formal" language. The definition of this language is based on a previously established ontology classicaly defining the concepts, relations and axioms of the domain.

B. Ontologies and requirements engineering

[9] describes the expected benefits but also the challenges of using ontologies in requirements engineering (RE) activities. This is exactly the basis of our approach. The main statement is that such approach needs the definition of three ontologies: (1) an **application domain** ontology, (2) a **requirements** ontology and (3) a requirements specification **document** ontology. The **application domain** ontology calls itself a double-utility ontology i.e. a **domain** ontology that defines the necessary concepts for all training in the domain, and an **application** ontology which is one defining the concepts specific to a given application or méthod. The **requirements** ontology is used for representing requirements and their various relationships, as well as the relationships between requirements and systems. Whereas the requirements specification **document** ontology is a documentary ontology. The present paper deals with the creation of an ontology the first type presented above.

C. Ontologies and railway systems / applications

In [10], the authors present an ontology creation work conducted during the FP6 InteGRail project [11]. They used the same tools as us (OWL, Protege) to modelize an ontology that permits to check network statement for infrastructure operators. Using the ontology, they combine the network statements of different countries in different formats and analyse them in a transparent way. They modelized the network using concepts like network node, network line, track section, track node. All these concepts permit to the authors to represent the railway network as an object graph. In our work we could reuse such concepts.

VIII. Conclusions and perspectives

In the present paper, we presented an experimental approach aiming at establishing an ontology of a complex domain like the ERTMS/ETCS railway control system. This development is mainly based on the study of a set of referential texts. As an example of the benefits we expect to obtain, we presented the enrichment of the ontology with the consideration of OSI standard levels to define precisely the concepts regarding the radio communication aspects of ERTMS.

SRS coverage: Since we focused on a first feasibility of the approach, the current coverage of the available texts by our ontology is obviously reduced. This work shall be improved in order to make our "ontological product" actually usable. It would be a painstaking work that could possibly take advantages on techniques (and related tools) such as automatic language processing. For example, the following step of our experiment may be the use of the GATE framework [12], since it provides ontological and also machine learning facilities.

Ontology quality: The quality of the ontology, viewed as a product, can be twofold: first, the quality of the embedded knowledge (as a semantic object); and second, the quality of the ontology itself (as a syntactic object).

The second point can be treated by the use of experience feedback from the elaboration of other ontologies and, particularly by taking into account the best practices of the domain sometimes identified and integrated into dedicated static analysis tools (like OOPS! [13]).

Table I gives the results of the evaluation of the ERTMS ontology by the tool OOPS!, in its current state. These results are barely correct because this "syntactic" aspect of the assessment has not been taken into account yet. For example, the pitfall with the worst score should be easily corrected simply by using the correct "annotation property" attribute for the definitions.

It is also possible to improve the overall quality (structure) of the current ontology by studying aspects like modularity ([14]), thus improving the decomposition and the potential reuse of the knowledge. A better modularity should also make easier the reuse of other related ontologies like testing ontologies, RE ontologies (as stated in section VII-B).

TABLE I. ONTOLOGY PITFALLS SCANNER OOPS!

	Pitfall	Cases
P04	Creating unconnected ontology elements	7
P05	Defining wrong inverse relationship	2
P08	Missing annotations	244
P11	Missing domain or range in properties	35
P13	Missing inverse relationships	33
P19	Swaping intersection and union	38
P21	Using a miscellaneous class	2
P22	Using different naming criteria in the ontology	ontology* (?)
P24	Using recursive definition	4

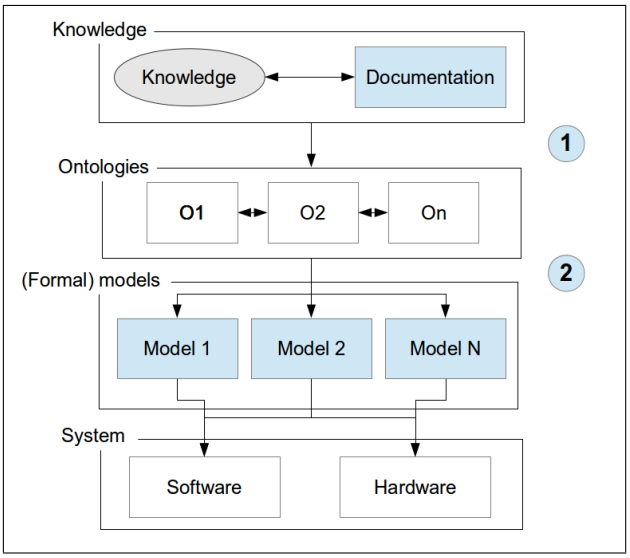


Fig. 6. Ontologies in the formalized developpement of safety critical systems

Usage of reasoners: The first point (knowledge quality) is essentially a matter of specific expertise to the considered field, but it can also be enhanced by the power of inference mechanisms used especially to detect semantic inconsistencies, incompleteness of relations, ...

All these criteria are not assessable by previous techniques (syntactic/structure level). As we study complex specification documents, it is even more important to implement these mechanisms earlier in the development process, so as to achieve a real "debugging" of the ontology before effective implementation of the system.

Linking ontology and external (formal) models: As a next step, when the ERTMS ontology will be rich enough to be usable, we will start to tackle the problem of deriving more concrete models (mainly formal ones). As described in figure 6, the current work (number one circled) deals with the analysis of available documentation and expert knowledge to derive one ontology (and probably several others in the future) which can be taken as a first step for an abstract formalization.

Formal methods are highly recommended for the development of safety-critical (railway) systems (cf. CENELEC 50128 Norm [15]). The ERTMS/ETCS is a system of this kind and, thus, a formalizable domain.

Indeed, in the openETCS project [16], a european large project involving the main actors of railway research, ERTMS/ETCS (semi-)formal models will be delivered (as well as the corresponding tool-chains). More than ten formalisms/approaches are studied. They range from ADA (the robust programming language), UML and/or sysML, to formal methods like SCADE, B or eventB, Petri nets, ...

The next step (number two circled) will be the derivation of more concrete models using available and well-known (semi-)formalisms like those used in the openETCS initiative. We intend to show that an initial formalization derived from an ontological conceptualisation will be helpful to define the architecture and the main properties for derived formal models.

Since the ontological support languages (OWL, SWRL, ...) used while elaborating our ontology are not too far from classical first order logic and set theory, one path to explore may be a model transformation from our ontology into formal specifications expressed within a "classical" formalism such as the B formal method [17].

Clearly, connecting our approach to the artefacts (formal models!) of the openETCS will be a real achievement.

A. Acknowledgements

The present research work is supported by the ICSIT (International Campus on Safety and Intermodality in Transportation) program and funded by the Nord Pas de Calais French Region and the ERDF (European Research and Development Funds). The authors gratefully acknowledge the support provided by these institutions.

REFERENCES

[1] D. Bjørner, "Rôle of domain engineering in software development—why current requirements engineering is flawed!" in *Perspectives of Systems Informatics*, ser. Lecture Notes in Computer Science, A. Pnueli, I. Virbitskaite, and A. Voronkov, Eds. Springer Berlin Heidelberg, 2010, vol. 5947, pp. 2–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11486-1_2

[2] A. S. Tanenbaum, *Computer Networks*. Prentice Hall, 1996.

[3] G. Thomas R., "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, pp. 199–220, 1993.

[4] J. Charlet, B. Bachimont, and R. Troncy, "Ontologies pour le web sémantique," *Action spécifique*, vol. 32, pp. 43–63, 2003.

[5] N. F. Noy and D. L. Mcguinness, "Ontology development 101: A guide to creating your first ontology," Online, 2001. [Online]. Available: http://www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html

[6] E. U. G. UNISIG, *System Requirements Specification (SRS) version 3.2.0*, E. R. Agency, Ed., 2012. [Online]. Available: http://www.era.europa.eu

[7] E. F. Barbosa, E. Y. Nakagawa, A. C. Riekstin, and J. Madonado, "Ontology-based Development of Testing Related Tools," 2008. [Online]. Available: http://www.labes.icmc.usp.br/moduloeducacional/publicacoes/SK06Ellen.pdf

[8] G. Bonifacio, P. Marmo, A. Orazzo, I. Petrone, L. Velardi, and A. Venticinque, "Improvement of processes and methods in testing activities for safety-critical embedded systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Flammini, S. Bologna, and V. Vittorini, Eds. Springer Berlin Heidelberg, 2011, vol. 6894, pp. 369–382.

[9] V. Castaneda, L. Ballejos, M. L. Caliusco, and M. R. Galli, "The use of ontologies in requirements engineering," *Global journal of researches in engineering*, vol. 10, no. Issue 6, Nov. 2010. [Online]. Available: http://www.engineeringresearch.org/index.php/GJRE/article/view/76/71

[10] S. Verstichel, F. Ongenae, L. Loeve, F. Vermeulen, P. Dings, B. Dhoedt, T. Dhaene, and F. D. Turck, "Efficient data integration in the railway domain through an ontology-based methodology," *Transportation Research Part C: Emerging Technologies*, vol. 19, no. 4, pp. 617–643, 2011.

[11] InteGRail Consortium, "Integrail, intelligent integration of railway systems." InteGRail Consortium, 2009. [Online]. Available: http://www.integrail.info

[12] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters, *Text Processing with GATE (Version 6)*, 2011. [Online]. Available: http://tinyurl.com/gatebook

[13] M. Poveda-Villalón, M. Suárez-Figueroa, and A. Gómez-Pérez, "Validating ontologies with oops!" in *Knowledge Engineering and Knowledge Management*, ser. Lecture Notes in Computer Science, A. Teije, J. Völker, S. Handschuh, H. Stuckenschmidt, M. d'Acquin, A. Nikolov, N. Aussenac-Gilles, and N. Hernandez, Eds. Springer Berlin Heidelberg, 2012, vol. 7603, pp. 267–281.

[14] C. Bezerra, F. Freitas, J. Euzenat, and A. Zimmermann, "ModOnto: A tool for modularizing ontologies," in *Proc. 3rd workshop on ontologies and their applications (Wonto)*, Salvador de Bahia, Brésil, Oct. 2008, p. No pagination., bezerra2008a INRIA-CNPq-OntoCompo; IST-NeOn. [Online]. Available: http://hal.inria.fr/hal-00793533

[15] CENELEC, "Railway applications - communications, signalling and processing systems - software for railway control and protection systems," 2011.

[16] openETCS, ITEA2 openETCS consortium, 2012. [Online]. Available: http://openetcs.org

[17] J.-R. Abrial, *The B Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

Verification of Scheme Plans using CSP||B

Philip James*, Faron Moller*, Hoang Nga Nguyen*, Markus Roggenbach*,
Steve Schneider†, Helen Treharne†, Matthew Trumble†, and David Williams‡
*Swansea University, Wales
†Department of Computing, University of Surrey
‡VU University, Amsterdam

***Abstract*—The paper presents a tool-supported approach to graphically editing scheme plans and their safety verification. The graphical tool is based on a Domain Specific Language which is used as the basis for transformation to a CSP||B formal model of a scheme plan. The models produced utilise a variety of abstraction techniques that make the analysis of large scale plans feasible. The techniques are applicable to other modelling languages besides CSP||B. We use the ProB tool to ensure the safety properties of collision, derailment and run-through freedom.**

I. Introduction

In a series of papers [11], [10], [12], [9], [14] we have been developing a new modelling approach for railway interlockings. This work has been carried out in conjunction with railway engineers drawn from our industrial partner. By involving the railway engineers from Invensys, we benefit twofold: they provide realistic case studies, and they guide the modelling approach, ensuring that it is natural to the working engineer.

We base our approach on CSP||B [16], which combines event-based with state-based modelling. This reflects the double nature of railway systems, which involves events such as train movements and – in the interlocking – state based reasoning. The formal models are by design close to the domain models. To the domain expert, this provides traceability and ease of understanding. The validity of this claim was demonstrated in particular in [11] where a non-trivial case study – a complex double junction – was provided, a formal model of which was understandable and usable by our industrial partners.

In [10], [14] we addressed how to *effectively* and *efficiently* verify safety properties within our CSP||B models. The properties of interest are collision, derailment and run-through freedom. To this end we developed a set of abstraction techniques for railway verification that allow the transformation of complex CSP||B models into less involved ones; we proved that these transformations are sound; and we demonstrated that they allow one to verify a variety of railway systems via model checking. The first set of abstractions reduces the number of trains that need to be considered in order to prove safety for an unbounded number of trains. Their correctness proof involves slicing of event traces. Essentially, these abstractions provide us with finite state models. The second set of abstractions simplifies the underlying track topology. Here, the correctness proof utilizes event abstraction specific to our application domain similar to the ones suggested by Winter in [18]. These abstractions make model checking faster.

Still present in our approach from the aforementioned papers was the need to write the formal models by hand. In [8] we described our OnTrack toolset[1], an open tool environment allowing graphical descriptions to be captured and supported by formal verification. This enables an engineer to visually represent the tracks and signals etc., within a railway network.

In this paper we continue the dissemination of our modelling approach which now also incorporates multi-directional tracks. We demonstrate that when changes are made to the models they are systematic and traceable; again this addition will be incorporated within our OnTrack tools.

The paper is organised as follows. In Section II we introduce our modelling language CSP||B so that we have the basis for discussing our workflow and provide examples. In Section III we describe the workflow for our CSP||B modelling approach and summarise where the different abstraction techniques fit into the workflow. In Section IV we introduce the modelling concepts of multi-directional travel and provide two illustrative examples. In Section V we put our work in the context of related approaches and finally conclude with future plans for the approach.

II. Background to CSP||B

The CSP||B approach [16] allows us to specify communicating systems using a combination of the B-Method [1] and the process algebra CSP (Communicating Sequential Processes) [6]. The overall specification of a combined communicating system comprises two separate specifications: one given by a number of CSP process descriptions and the other by a collection of B machines. Our aim when using B and CSP is to factor out as much of the "data-rich" aspects of a system as possible into B machines. The B machines in our CSP||B approach are classical B machines, which are components containing state and operations on that state. The CSP||B theory [16] allows us to combine a number of CSP processes *Ps* in parallel with machines *Ms* to produce *Ps* $\parallel$ *Ms* which is the parallel combination of all the controllers and all the underlying machines. Such a parallel composition is meaningful because a B machine is itself interpretable as a CSP process whose event-traces are the possible execution sequences of its operations. The invoking of an operation of

[1]OnTrack available for download from http://www.csp-b.org.

a B machine outside its precondition within such a trace is defined as divergence [13]. Therefore, our notion of consistency is that a combined communicating system *Ps* $\parallel$ *Ms* is *divergence-free*. We do not consider deadlock-freedom in this paper as it is concerned with liveness, and the focus of the paper is on safety.

A B MACHINE clause declares a machine and gives it a name. The VARIABLES of a B machine define its state. The INVARIANT of a B machine gives the type of the variables, and more generally it also contains any other constraints on the allowable machine states. There is an INITIALISATION which determines the initial state of the machine. The machine consists of a collection of OPERATIONS that query and modify the state. Besides this kind of machine we also define static B machines that provide only sets, constants and properties that do not change during the execution of the system.

The language we use to describe the CSP processes for B machines is as follows:

$$P ::= e?x!y \rightarrow P(x) \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ end} \mid N(exp) \mid P_1 \parallel P_2 \mid P_1 \,{}_A\|_B\, P_2 \mid P_1 \,|||\, P_2$$

The process $e?x!y \rightarrow P(x)$ defines a channel communication where x represents all data variables on a channel, and y represents values being passed along a channel. Channel e is referred to as a *machine channel* as there is a corresponding operation in the controlled B machine with the signature $x \longleftarrow e(y)$. Therefore the input of the operation y corresponds to the output from the CSP, and the output x of the operation to the CSP input. Here we have simplified the communication to have one output and one input but in general there can be any number of inputs and outputs. The other CSP operators have the usual CSP semantics.

In this paper we omit a detail discussion of the semantic models used for reasoning of CSP||B models. In [14] we discuss that the traces models is enough to deal with the safety properties of railway interlockings.

III. VERIFICATION WORKFLOW

Figure 1 shows the workflow that we employ in our methodology. It makes use of two tools: OnTrack and the ProB model checker [15]. Initially, a user draws a *Track Plan* using the graphical front end in the OnTrack tool. Then the first transformation, *Generate Tables* leads to a *Scheme Plan*, which is a track plan and its associated control and release tables. Control tables contain information about when routes can be granted and release tables contain information about when points can be released, see [11] for details. Note in the paper we typically refer to tracks as being both linear tracks or points. Track plans and scheme plans are models formulated relative to our railway domain-specific language (DSL) meta-model [8]. A scheme plan is the basis for subsequent workflows that support its verification. Scheme plans can be captured as formal specifications. The simplest transformation, indicated by the *Transformation* dashed arrow, is to produce one *Formal specification* that is a faithful representation of the scheme plan. This transformation is a mapping from the railway DSL meta-model to the CSP||B meta-model and its subsequent representation as CSP||B script files that can be inputted into ProB. This automated transformation makes use of the finitisation theory in order to be able to perform bounded model checking of the formal specification [10], [7]. The finitisation theory allows us to reduce the problem of verifying of scheme plans for safety (i.e., freedom from collision, derailment, and run-through) for any number of trains to that of a two-train scenario.

Nonetheless, even with the examining a reduced number of trains the formal specifications of realistic examples will inevitably contain too many states for safety analysis. Thus, our methodology enables us to carry out two forms of abstraction on a scheme plan:

(1) Covering Abstraction supports the decomposition of a scheme plan with a set of smaller sub-scheme plans. Any particular track in a scheme plan has a 'zone of influence': the other tracks which need to be considered to see what will happens on that track (e.g., when routes including it are enabled, when trains are approaching it, etc.). In particular, we only need to look at the zone of influence in order to see if a collision is possible on that track. To analyse if a collision, derailment or run-through is possible on that track, it is enough just to analyse the behaviour of trains within the zone of influence. We can do this for all the tracks, in each case just analysing for collisions, derailment or run-through within its zone of influence. This is called a covering. In general each zone of influence is much smaller than the overall track plan, so the analyses will be much quicker, and in practice can be done efficiently.

(2) Topological Abstraction supports the collapsing of tracks of a scheme plan to minimise the number of superfluous tracks in a plan, i.e., ones which do not impact on safety. Thus, for a particular track plan we take a sequence of tracks, and think of them as one single track. We do this for a number of sequences of tracks along the way. It is a topological abstraction if we can match moves around the original track plan with moves around the smaller one, so changes such as routes being enabled, points being released, trains being on particular routes, points being set, trains being at lights must still match for this collapsing to be a topological abstraction. If this is true then it means that we can analyse the behaviour of trains on the smaller scheme plan (which is easier because there are fewer positions to consider) and the results that we get will still be true for the original larger scheme plan.

We have proved the soundness of these abstractions in [10], [7]. In our methodology we first apply covering abstraction to generate sub-scheme plans and then apply topological abstraction to each of them. Using these abstractions we follow the *Abstraction* vertical workflow from the scheme plan to produce one or more *Minimised abstract sub-scheme plan(s)*. One or more such plans may be produced because as we shall see in our examples, in Section IV, it may not always be possible to perform covering, and in which case the only abstraction that

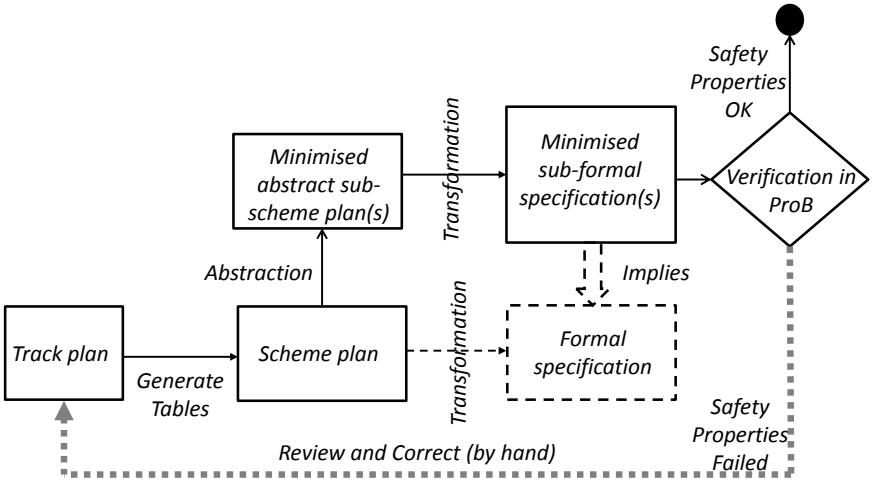


Fig. 1. CSP||B modelling and verification workflow.

may yield a reduction in the number of tracks in the plan will be topological abstraction. Applying these abstractions is done at the DSL level and is independent of the formalism being used to represent the abstract CSP||B specification. Currently, the covering abstraction is not fully automated but is ongoing development work within the OnTrack tool.

Following abstraction (top left box on the diagram) the *Transformation* workflow, described earlier, can be applied to the minimised abstract sub-scheme plans to produce corresponding sub-formal specifications. All of the transformations that are performed by the OnTrack tool are validated via manual review. The verification of all of these sub-formal specifications implies the safety of the formal specification, as illustrated by the *Implies* arrow workflow; this result has been formally proved [10], [7].

Once OnTrack produces the sub-formal specifications they are all systematically verified using the ProB model checker to ensure that the models are collision- and derailment-free and contain no run-throughs. Successful checks verify that the safety properties hold for the particular scheme-plan. The workflow has the potential for round-trip engineering where the counter examples produced from unsuccessful model checking are automatically fed back into the OnTrack tool. This has not, as yet, been incorporated into the tool but it would provide an improved tool-supported workflow; this is illustrated using the dotted *Review and Correct* arrow on the workflow.

IV. Modelling of Multi-directional Examples of CSP||B railway models

In this section we provide details of the architecture of the formal specifications that are produced by the OnTrack tool. The architecture of a CSP||B specification presented in [11] is restated in Figure 2. The centralised control logic is represented in the *Interlocking* machine, whereas the train

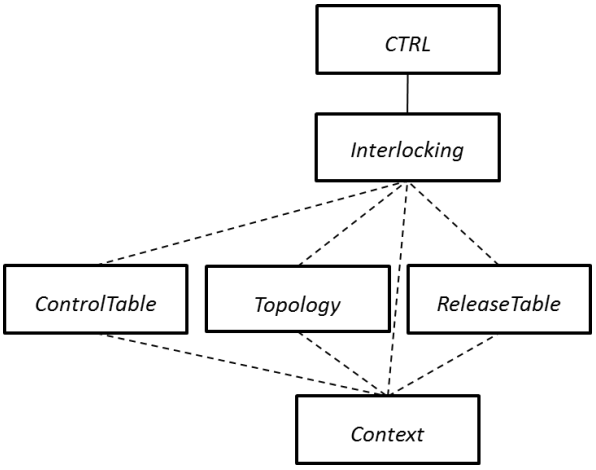


Fig. 2. CSP||B Architecture.

behaviour is controlled by CSP processes defined in the *CTRL* script. These process and machine synchronise on common events. Their definitions are independent of any particular scheme plan but contain the state of the railway interlocking model which changes as these events occur. The definitions are supported by generic domain definitions contained in the following stateless machines: *Topology*, *ControlTable*, *ReleaseTable* and *Context*. The sets, relations and functions in these stateless machines are automatically instantiated for a particular scheme plan. The definitions of the types in the *CTRL* script are also automatically instantiated to match the B instantiations. In the next sections we illustrate some aspects of the CSP processes and machines via examples [2] and focus on how multi-directional travel of trains on tracks is modelled.

[2]Examples available for download from http://www.csp-b.org.

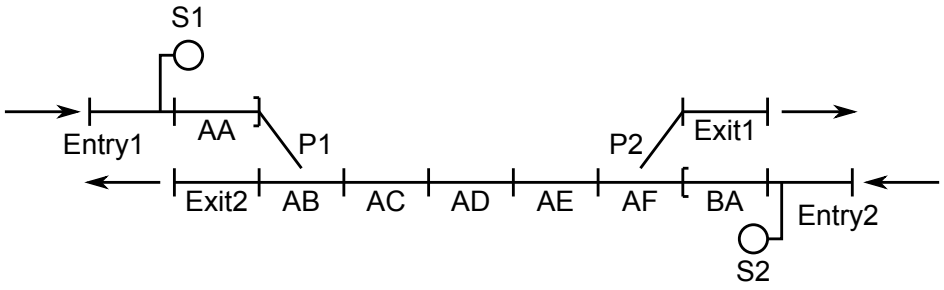


Fig. 3. Track plan for the tunnel example

```
1  TRAIN_CTRL(t, pos) = ...
2  □ pos ∉ EXIT ∧ pos ∉ SIGNALHOMES &
3        move!t.pos?newp → TRAIN_CTRL(t, newp)
4  □ ...
```

Fig. 4. Fragment of the CSP control process for trains.

A. Tunnel Example

Consider the track plan in Figure 3 where tracks *AB*, *AC* and *AD* are bi-directional tracks. For route $R1$ associated with signal $S1$ their direction is left to right, whereas for route $R2$ associated with signal $S2$ their direction is right to left. The CSP process that controls the movement of trains is *TRAIN_CTRL*. Figure 4 illustrates the fragment of it controlling the movement of a train from a track that is neither an *exit* one or one which has a signal on it. The *move* event is parameterised with the train identifier t and its current position p. This event is a synchronisation with a *move* B operation which returns its new position *newp*. Therefore, moving from track *AC* to *AD* corresponds to the event *move.t.AC.AD* for a particular train t.

Note, there is no information in the CSP event that corresponds to the direction of travel. All this information is contained in the *Topology* machine and used in the *move* operation within the *Interlocking* machine. In the *Topology* machine there are three relations which define the direction of tracks. For example, the relation *direction* shown in Figure 5 shows that the model needs to contain details of the way tracks are connected together, and this is explicitly done via the notion of identified *connectors* — the glue between tracks and points.

```
1  direction ∈ TRACK ↔ CONNECTOR * CONNECTOR ∧
2    direction = {...,
3      AA ↦ (C1, C2), ...,
4            AC ↦ (C3, C4), AC ↦ (C4, C3),
5            AD ↦ (C4, C5), ...}
```

Fig. 5. Fragment of the *direction* relation from *Topology*.

As we saw above the notion of a train's position in the CSP was captured using two parameters (t, pos). In the INVARIANT of the *Interlocking* machine a similarly named function *pos* also includes information about the connectors, as shown in Figure 6. In its INITIALISATION $pos := \emptyset$ since there are no trains on the tracks. The *move* operation updates the track and connectors related to train t in *pos* each time the train moves. (In earlier papers, e.g., [11], *pos* was simply a partial function between trains and tracks and *direction* was not required.)

```
1  pos ∈ TRAIN ⇸ ALLTRACK*
2        (ALLCONNECTOR * ALLCONNECTOR)
```

Fig. 6. *pos* function from *Interlocking*.

In addition to B operations which define the behaviour of movement, granting and releasing of route requests the OnTrack tool automatically produces B operations to support the verification of safety properties. Three B operations are produced, *collision*, *derailment* and *run-through*. Collision is encoded as follows:

```
1  collision =
2  SELECT
3    ∃t1, t2 ∈ TRAIN ∧ t1 ≠ t2 ∧
4    t1 ∈ dom(pos) ∧ t2 ∈ dom(pos)
5    (dom(pos(t1)) − (EXIT ∪ ENTRY)) ∩
6    (dom(pos(t2)) − (EXIT ∪ ENTRY)) = ∅
7  THEN skip
8  END;
```

Here collision is detected when two different trains t_1 and t_2 occupy the same track segment (different from the *EXIT* and *ENTRY* tracks). The collision condition will be enabled when the two trains are at the same position.

Collision freedom can then be established by model checking the validity of the following CTL formula:

$$AG(not(e(collision)))$$

This formula is false if *collision* is enabled. In the CTL variant of PROB *AG*, stands for "on all paths it is globally true that", and $e(a)$ stands for "event a is enabled". To achieve this the engineer would take the formal specification produced by OnTrack load it into the ProB tool and perform this check. A total of 1,516 distinct states were examined in order to determine that no collision was possible. Our methodology currently requires us to do this loading by hand but automating this as a batch process for all the safety properties could easily be done.

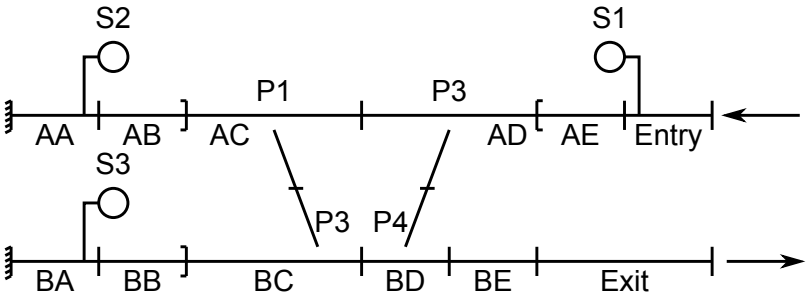


Fig. 7. Track plan for the buffer example

B. Buffer Example

Our next example is also multi-directional as shown in Figure 7. Interestingly, track *BC* has three directions, i.e., $\{BC\} \lhd direction = \{(C12, C11), (C11, C12), (C7, C12)\}$, where $C7$ is the connector between tracks *AC* and *BC*, $C11$ is between *BB* and *BC*, and $C12$ is between *BC* and *BD*, respectively.

It also serves to illustrate how additional complexity can easily be traced within a formal specification. We model the behaviour of buffers, i.e., tracks where trains can turn around; in our example the buffers are *AA* and *BA*. Two routes are associated with signal $S1$, i.e., route $R1A$ is associated with *AE*, *AD*, *AC*, *AB* and *AA* and $R1B$ is associated with *AE*, *AD*, *BD*, *BC*, *BB* and *BA*. Thus, when a train is on route $R1A$ and is on track *AA* it can change direction and then follow route $R2$ which is associated with signal $S2$. Similarly, for route $R3$ associated with signal $S3$.

This additional behaviour requires three additions to the CSP processes and B machines:

- The additional definition of $BUFFER = \{AA, BA\}$ in the *Context* machine and similarly in the CSP types.
- A new *changeDirection* operation as shown in Figure 8. The purpose of this operation is to simply modify the direction of the connectors for the particular buffer track on which the train t currently resides. Hence, changing the direction of train t on track *AA* means changing the maplet $(t \mapsto AA, (C1, C0))$ to $(t \mapsto AA, (C0, C1))$ within the *pos* function. This means that we can leave the *move* operation unchanged.
- Within the CSP, rather than disturb the existing processes, we define a new process, $BUFFER_P(b, t)$ in Figure 9 which defines that after a train moves onto the buffer track b it must change direction before it can move off it. In the model there will be a separate buffer process for each buffer and they are independent of each other. These new processes are combined to reformulate the overall CSP processes contained in the *CTRL* script.

The state space required by ProB to model check the safety properties for the formal specification of the Buffer example was 18,510 states, significantly more than in the tunnel example. In Section III we noted that it may not always be feasible to model check a complex scenario but our methodology supports the systematic generation of all the

```
1  changeDirection(t, currp) =
2  PRE t ∈ TRAIN ∧ t ∈ dom(pos)∧
3        {currp} = dom({pos(t)}) ∧ currp ∈ BUFFER
4  THEN
5    movedPoints := {} ||
6    LET(track, d) BE (track, d) = pos(t) IN
7      LET(d1, d2) BE (d1, d2) = d IN
8        pos(t) := (track, (d2, d1))
9      END
10   END
11 END;
```

Fig. 8. *changeDirection* method from *Interlocking*.

```
1  BUFFER_P(b, t) = move!t?p!b → changeDirection.t.b
2                              → move.t.b?newp → BUFFER_P(b, t)
```

Fig. 9. $BUFFER_P$ process in *CTRL*.

sub-scheme plans for a particular scheme plan. The track plan for one of the sub-scheme plans of the buffer example is shown in Figure 10. It illustrates the plan for the track *AC* constructed using the covering abstraction. The highlights from this plan are as follows:

- The point *BC* in the overall buffer example can now be considered as an *exit* track and after which we do not need to consider the behaviour of subsequent linear tracks and points. The reason being is that all that needs to be captured is what happens to the state when a train moves off the point *AC* and that this can be represented using a simple linear track rather than a point.
- The point *BD* is similarly converted to an *exit* track.
- The current version of our covering technique has not considered the impact of buffers on the abstraction of the scheme plans. Therefore, we must include them in the zone of influence. Therefore, both tracks *AA* and *AB* retain their bi-directional properties in the sub-scheme plan. We shall of course examine in future work whether such tracks can be further reduced.
- Notice also that we need not consider the path along *Entry*, *AE*, *AD*, *BD*, *BC*, *BB*, *BA* because it does not belong to the zone of influence as it does not contain the track *AC*, but of course *Entry*, *AE* and *AD* are included because they are on the normal route $R1A$ and *BD* is

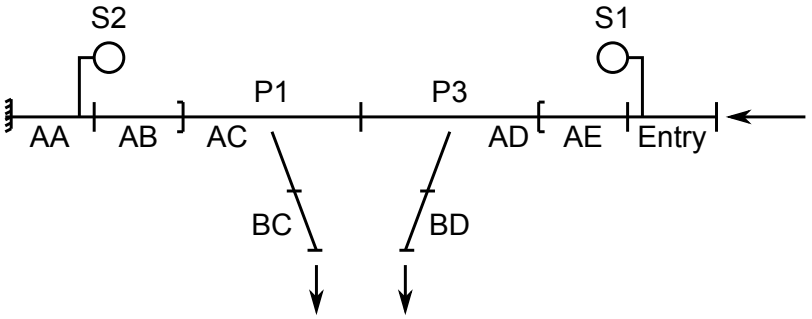


Fig. 10. Sub-track plan for track *AC* of the buffer example

included for the above reason.

Running the formal specification of the sub-scheme plan for *AC* through ProB gives a state space of 3,995 compared to 18,510 states for the full specification. We have also verified that the three important safety properties hold for this sub-scheme plan. Methodologically, we would then be required to run all the sub-scheme plans through ProB and by appealing to our theoretical results we would conclude that the overall buffer example from Figure 7 preserves the safety properties.

V. Related work

To put our work into context we must first clarify that railway verification falls into two categories: the verification of railway designs prior to their implementation and the verification of the implementation descriptions themselves. Our work is in the first area. A comparison using different model checkers in the analysis of control tables has been conducted by Ferrari *et al.* [3] and falls into the first category. Winter in a recent paper [17] considers different optimising strategies for model checking using NuSMV and demonstrates the efficiency of their approach on very large models. These analyses also fall into the first category but the models are flat in structure compared to our models as they are defined in terms of boolean equations and do not focus on providing behavioural models. The analysis of interlocking tables (cf. control tables) by Haxthausen [4] also falls into the first category and is supported by automated tools that generate the models. Cimatti *et al.* [2] also have had considerable success using NuSMV but their analysis is focussed on the implementation descriptions.

VI. Conclusion

In this paper we provided an overview of our methodology that uses the OnTrack tool to provide a graphical front-end for the automatic generation of formal specifications. The formal specifications are then separately model checked using the ProB tool. We described the architecture of a CSP||B formal specification of a scheme plan giving details of the new aspects that allow the modelling of multi-directional travel. We appreciate the absolute necessity to include these aspects in our CSP||B formal specifications and recognise that the majority of the related work includes such detail, for example [4]. Our aim by demonstrating its inclusion incrementally was to show the robustness of the CSP||B architecture and the ease by which new modelling aspects can be included. Similarly, additional development of the OnTrack tool-support can also be achieved incrementally. We are currently completing the implementation of the covering abstractions and the integration of the output from ProB model checking with OnTrack in order to provide round-trip engineering to the graphical editor. This will mean that the engineer is not required to manipulate the formal specifications when safety properties are violated. Instead, the engineer will be able to change a graphical scheme plan, re-generate the formal specifications and re-run the model checking in order to verify that the amended scheme plan preserves safety (i.e., freedom from collision derailment and run-through).

Heitmeyer in [5] discusses the importance of complete abstractions. Our abstractions are sound. It is future theoretical work to investigate if completeness can be established.

Acknowledgment

Thanks to S. Chadwick and D. Taylor from the company Invensys Rail for their support and encouraging feedback.

References

[1] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.
[2] A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltsev. Formal verification and validation of ERTMS industrial railway train spacing system. In *CAV*, pages 378–393. Springer, 2012.
[3] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model checking interlocking control tables. In *FORMS/FORMAT*, pages 107–115, 2010.
[4] A. Haxthausen. Automated generation of safety requirements from railway interlocking tables. In *ISoLA (2)*, volume 7610 of *Lecture Notes in Computer Science*, pages 261–275, 2012.
[5] C. L. Heitmeyer, J. Kirby, B. G. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Software Eng.*, 24(11):927–948, 1998.
[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[7] P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Techniques for modelling and verifying large scale railway interlockings (under consideration), 2013.
[8] P. James, M. Trumble, H. Treharne, M. Roggenbach, and S. Schneider. OnTrack: An open tooling environment for railway verification. In *Proceedings of NFM'13: Fifth NASA Formal Methods Symposium*, 2013.
[9] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Combining event-based and state-based modelling for railway verification. Technical Report CS-12-02, University of Surrey, 2012.

[10] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Defining and model checking abstractions of complex railway models using CSP||B. In *Proceedings of HVC'12: Eighth Haifa Verification Conference (to appear in Springer Lecture Notes in Computer Science)*, page 16 pages, 2012.

[11] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H Treharne. Railway modelling in CSP||B: The double junction case study. *Electronic Communications of the EASST*, 53:15 pages, 2012.

[12] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Using ProB and CSP||B for railway modelling. In *Proceedings of IFM'12 and ABZ 2012 Posters and Tool demos session*, pages 31–35, 2012.

[13] C. Morgan. Of wp and CSP. *Beauty is our business: a birthday salute to E. W. Dijkstra*, pages 319–326, 1990.

[14] F. Moller P. James, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. On modelling and verifying railway interlockings: Tracking train lengths. Technical Report CS-13-03, University of Surrey, 2012.

[15] The ProB animator and model checker (ProB 1.3.6-final). `http://www.stups.uni-duesseldorf.de/ProB`. Accessed: 01/05/2013.

[16] S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.

[17] K. Winter. Optimising ordering strategies for symbolic model checking of railway interlockings. In *ISoLA (2)*, volume 7610 of *Lecture Notes in Computer Science*, pages 246–260, 2012.

[18] K. Winter and N.J. Robinson. Modelling large railway interlockings and model checking small ones. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 309–316. Australian Computer Society, Inc., 2003.

Applied Bounded Model Checking for Interlocking System Designs

Anne E. Haxthausen
DTU Compute
Technical University of Denmark
Email: aeha@dtu.dk

Jan Peleska
Department of Mathematics and Computer Science
Universität Bremen, Germany
Email:jp@informatik.uni-bremen.de

Ralf Pinger
Siemens AG, Braunschweig, Germany
Email:Ralf.pinger@siemens.com

***Abstract*—In this article the verification and validation of interlocking systems is investigated. Reviewing both geographical and route-related interlocking, the verification objectives can be structured from a perspective of computer science into (1) verification of static semantics, and (2) verification of behavioural (operational) semantics. The former checks that the plant model – that is, the software components reflecting the physical components of the interlocking system – has been set up in an adequate way. The latter investigates trains moving through the network, with the objective to uncover potential safety violations. From a formal methods perspective, these verification objectives can be approached by theorem proving, global, or bounded model checking. This article explains the techniques for application of bounded model checking techniques, and discusses their advantages in comparison to the alternative approaches.**

***Index Terms*—railway control systems, interlocking systems, formal methods, bounded model checking, temporal logic**

I. Introduction

Formal methods have been applied for years in the railway domain and reached a level that enables the compilation of the body of knowledge in the form of an engineering handbook (in the style of [1]), recording case-based "best practices". To this end, this paper contributes knowledge concerning verification and validation (V&V) of interlocking system designs. First we outline the state-of-the-art of V&V tasks and formal methods for performing them. Then techniques for applying one of these methods (bounded model checking) are explained in more detail.

A. Interlocking V&V – State-of-the-art

Software controlling interlocking systems has to be verified on two levels. The first level focuses on the correctness of configuration data specifying how the topology of the railway network controlled by the interlocking system is reflected by re-usable software objects, their interfaces, and their instantiation data. Correctness of the configuration data ensures that the software has adequate control over the electromechanical components of the physical interlocking system. In terms of computer science, this is a check of static semantics. The second verification level investigates the safety of trains passing through the controlled network area. The verification objective is to prove the absence of hazardous situations in the network, provided that all trains follow the restrictions (signals, speed limitations) imposed by the interlocking system. This corresponds to a property check of the interlocking systems' behavioural semantics.

Interlocking systems are designed according to different paradigms [2, Chapter 4]. Two of the most widely used ones are (a) *geographical interlocking systems* and (b) *route-based interlocking systems* using interlocking tables. For design type (a), routes through the railway network can be allocated dynamically by indicating the starting and destination points of trains intending to traverse the railway network portion controlled by the interlocking system under consideration. In the original technology, electrical relay-based circuits were applied, whose elements and interconnections where designed in one-to-one correspondence with those of the physical track layout. The electric circuit design ensured dynamic identification of free routes from starting point to destination, the locking of points and setting of signals along the route, as well as on neighbouring track segments for the purpose of flank protection. In today's software-controlled electronic interlocking systems, instances of software components "mimic" the elements of the electric circuit. Typically following the object-oriented paradigm, different components are developed, each corresponding to a specific type of physical track element, such as points, track sections associated with signals, and others with axle counters or similar devices detecting trains passing along the track. Similar to connections between electric circuit elements, instances of these software components are connected by communication channels reflecting the track network. The messages passed along these channels carry requests for route allocation, point switching and locking, signal settings, and the associated responses acknowledging or rejecting these requests. The software components are developed for re-use, so that novel interlocking software designs can be realised by means of configuration data, specifying which instances of software components are required, their attribute values, and how their communication channels shall be connected. The geographical approach to interlocking system design induces a separate verification and validation (V&V) step which is called *data validation*. Its objective is to check whether the instantiation of software components is complete, each component is equipped with the correct attribute values, and whether the channel interconnections are adequate. The data validation objectives are specified by means of rules,

and the rules collection is usually quite extensive (several hundred), so that manual data validation is a cumbersome, costly, and error-prone task. Moreover, the addition of new rules often required expensive extensions of manually programmed checking software. Data validation investigates only the static semantics of the network of software components. A second V&V step is required to check whether the design will ensure the safety properties required, so that – at least under certain boundary conditions stating that train engine drivers have to respect signals and speed restrictions, as far as not automatically enforced by the underlying technology – trains moving concurrently through the railway network are protected against derailing and collisions.

Route-based interlocking (system type (b)) is less flexible than geographical interlocking, since it fixes all train routes through the railway network a priori, using route tables specifying the sequences of track segments to be allocated for each route. This loss of flexibility is compensated by the advantage that configuration data is considerably simpler. The route table is complemented by interlocking tables specifying the point positions and signal states to be enforced when allocating routes. The interlocking tables fix these positions both for the track elements which are part of the actual route, and the elements which are outside the route, but contribute to its safety by guaranteeing flank protection. Finally, a route conflict table identifies the routes which may never be simultaneously allocated, due to utilisation of common track elements [3]. Route-related interlocking offers simpler means for data validation, since the control software does not need to to be based on communicating software instances related to each track element. Instead, a control algorithm monitors a dynamic plant model (each track element with its free/occupied status, and the locked/unlocked states of points). Route allocation decisions can made by means of these element states and their compatibility with the interlocking table restrictions. Data validation is only concerned with choosing the proper software components (e.g., the correct types of signals and points), and their consistency with the physical network. V&V of the dynamic behaviour now has the objective to verify both the correctness of the control algorithm and the correctness of the interlocking tables. Even in presence of a completely correct algorithm, a safety violation may occur if these tables are not adequately specified; e.g., if a conflict between two routes has not been properly documented in the tables. As a consequence, the data validation activities concerning static semantics of the software components is simpler and less critical than in the case of geographical interlocking systems, but only V&V of the dynamic behaviour can verify the crucial safety properties of the interlocking tables.

B. State-of-the-art Formal Methods for Interlocking V&V

The European CENELEC standards applicable for the development of software in railway control systems require the application of formal specification and design models and formalised, justified V&V activities to be performed for software of the highest criticality, as applicable for interlocking systems [4]. The objective of such formalizations is to ensure that potential safety breaches caused by invalid configuration data or erroneous control algorithms can be identified in a systematic way. If formal methods application can also be "mechanised" by means of suitable tools, it contributes to the efficiency of V&V for interlocking system designs in a considerable way. As of today, three methods are applied for formal interlocking V&V: formal verification by theorem proving, by global model checking, or by bounded model checking (BMC). Each of these methods depends on the existence of models describing the static semantics of the interlocking systems, and their dynamic behaviour in combination with trains traversing the railway network.

While – just like theorem proving – global model checking may result in complete correctness proofs of data correctness and safety properties, experience (see for instance [5]) has shown that complex interlocking systems cannot be verified by means of global model checking, since this would lead to state explosions for all but the simplest interlocking systems. In contrast to this, bounded model checking investigates model properties in the vicinity of a given state only, and can therefore be applied to models of considerable size. In this contribution we describe first how BMC is applied to data validation. This is performed by checking the compliance of the data with correctness rules that may be expressed formally by some temporal logic. Next, for the verification of safety properties, BMC can be combined with inductive reasoning, and again, this results in a global proof of the desired safety properties. The bounded model checking techniques to be applied are sufficiently mature today to be applied in an industrial context.

C. BMC as Best Practice for Interlocking V&V

The bounded model checking solution to data validation is explained for geographical interlocking systems, since there the requirements for this validation are far more complex than for route-related interlocking. We describe how the software components instantiated according to the given configuration data can be formalised by means of a Kripke Structure whose state space is given by the software component instances, where the transition relation is induced by the communication channels connecting neighbouring objects, and the labelling function specifies the attributes associated with each instance. It is explained how typical pattern of data validation rules can be expressed by means of Linear Temporal Logic (LTL) including existential quantification of specific variable values. A trace of states fulfilling such a formula identifies a witness for a *violation* of the validation rule. Application of LTL model checking allows for easy extendability of the rule base, by simply adding new LTL formulas representing violations of the new rules. No further software extensions are required, as long as a sufficiently powerful bounded model checker for LTL exists. We further describe how the BMC approach can be rightfully applied, because each data validation rule only applies to a finite trace through the Kripke structure (while LTL property checking in general refers to infinite

computations). A bounded LTL property checking algorithm is sketched which can be efficiently applied for performing the data validation activities.

In [6] we have described a formal, model-driven method for efficient development and verification of product lines of reconfigurable route-related interlocking systems. This method is based on many years of research of which the most recent publications include [3] and [7], [8]. According to this method the development and verification of an interlocking system should be made in a number of steps including the following ones: (1) Specify application-specific parameters in a domain-specific railway language, and (2) from the domain-specific specification, generate a formal, behavioural model of the interlocking system and formal specification of the required safety properties. This generation should be fully automated by tools developed for the purpose. For this setting we describe how BMC may be applied in combination with inductive reasoning, in order to verify global safety properties of the interlocking system software and configuration data generated from these models. This combination of BMC and induction is well-established today in many domains, and it is known to scale up for complex "real-world" applications.

D. Related Work

An overview of trends in formal methods applications to railway signalling can be found in [9], [10]. Many other research groups have been using model-checking for the verification of interlocking systems. In [5] a systematic study of applicability bounds of the symbolic model-checker NuSMV and the explicit model checker SPIN showed that these popular model checkers could only verify small railway yards. Several domain-specific techniques to push the applicability bounds for model checking interlocking systems have been suggested. Here we will just mention some of the most recent ones. In [11] Winter pushes the applicability bounds of symbolic model checking (NUSMV) by optimizing the ordering strategies for variables and transitions using domain knowledge about the track layout. Fantechi suggests in [12] to exploit a distributed modelling of geographical interlocking systems and break the verification task into smaller tasks that can be distributed to multiple processors such that they can be verified in parallel. In [13], it is suggested to reduce the state space using abstraction techniques reducing the number of track sections and the number of trains.

For the alternative approach to interlocking V&V based on theorem proving, the B-Method and its variants, such as Event-B, seem to be the formal methods most strongly favoured for railway control applications in Europe. The formal verification of behavioural properties is described, and the methods' applicability on an industrial scale has been established, for example, in [14]. In [15], [16], the application of Event-B to data validation is described. Further verification approaches using theorem proving have been based on the RAISE method, as described in [17].

An introduction into LTL can be found in [18]. The existential quantification operator for LTL, which plays a crucial role in our concept of automated data validation, has been originally introduced in [19]. Its adaptation to finite trace semantics has been performed by the authors. The original semantics and algorithms for verifying LTL formulas against finite trace segments have been devised in [20], [21]. On these finite segments only a subclass of LTL formulas can be verified, this class has been identified in [22]. Fairness properties, for example, which can be expressed in the complete LTL with infinite computations as models, are not part of this class. Our data validation properties, however, as well as the safety properties to be fulfilled by the behavioural interlocking system semantics, are all part of the so-called *Safety LTL* subset which is expressible on finite trace segments.

E. Paper Overview

Sections II and III describe our methods for data validation and for verifying system safety, respectively. In Section IV, the presented methods are discussed.

II. Data Validation

A. Kripke Structure Encodings of Static Plant Model

As sketched above, the software controlling geographical interlocking systems consists of instances communicating over channels, each instance representing a physical track element in the plant model. A subset of these channels – called primary channels in the following – reflect the physical interconnection between neighbouring track elements which are part of possible routes, to be dynamically allocated when a request for traversal from some starting point to a destination is given (Fig. 1). Other channels – called secondary channels – connect certain elements s_1 to others s_2, such that s_1 and s_2 are never neighbouring elements on a route, but s_2 may offer flank protection to s_1, when some route including s_1 should be allocated. Since geographical interlocking is based on request and response messages, each channel for sending request messages from some instance s_1 connected to an instance s_2 is associated with a "response channel" from s_2 to s_1. Main channels are subsequently denoted by variable symbols a, b, c, d, while auxiliary channels are denoted by e, f, g, h.

All software instances are associated with a unique id. Depending on the track element type they are representing in the plant model, software instances carry an element type t. Depending on the type, a list of further attributes $a_1, \ldots, a_k$ may be defined for each software instance. By using a default value 0 for attributes that are not used for a certain component type, each element can be associated with the same complete list of attributes, where the ones which are not applicable are set to 0. Each valuation of a channel variable contains either a default value 0, meaning "no connection on this channel", or the instance identification $id > 0$ of the destination instance of the channel.

We will now formalise the static design of geographical interlocking systems as a Kripke Structure $K = (S, S_0, R, L, AP)$, with state space S, set of initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$ and labelling function $L : S \to 2^{AP}$, where AP is a set of atomic

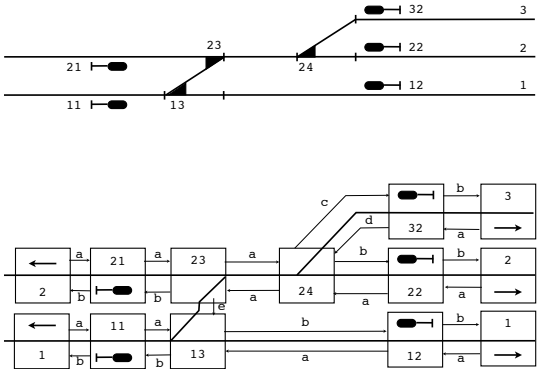


Fig. 1. Physical layout, associated software instances and channel connections.

propositions and 2^{AP} denotes its power set [18]. To this end, define a set V of variable names as introduced above, $V = \{id, t, a, b, c, d, e, f, g, h, a_1, \ldots, a_k\}$. The state space S consists of one valuation function $s : V \to \mathbf{N}_0$ for each software component. Each function maps the variables to integers identifying the associated software component (id is mapped to its unique id, t to its type, etc.). The set of initial states S_0 is defined to be the set of all states S. This allows us to start data validations at arbitrary track elements. The transition relation R defines each instance s_2 reachable from some instance s_1 via any of the channels $a, \ldots, h$ to be a possible post-state of s_1.

$$R \;=\; \{(s_1, s_2) \mid s_1(v) = s_2(id) \wedge v \in \{a, \ldots, h\}\}$$

The set of atomic propositions AP is defined as the collection of all propositions stating equality of some attribute $v \in V$ to one of its possible values, $AP = \{v = \xi \mid v \in V \wedge \xi \in \mathbf{N}_0\}$. The labelling function L maps each state s to the set $L(s)$ of propositions which hold true in s, that is, $\forall s \in S : L(s) = \{v = s(v) \mid v \in V\}$.

Now the violation of any data validation rule may be defined as a LTL formula specifying witnesses of such an unwanted sequence of neighbouring elements. This will be illustrated in the following by a collection of validation examples.

B. LTL Syntax

The LTL formulas specifying witnesses for rule violations use symbols from V as free variables. The atomic propositions involved may consist of arithmetic expressions and comparison operators $=, <, >, \leq, \geq, \neq$. The valid LTL formulas are constructed according to the following rules.

- Every atomic proposition is a LTL formula.
- If φ, ψ are LTL formulas, then[1] $\neg\varphi$, $\phi \wedge \psi$, $\phi \vee \psi$, $(\exists b : \varphi)$, $\mathbf{F}\varphi$, $\mathbf{G}\varphi$, $\mathbf{X}\varphi$, $(\varphi \mathbf{U} \psi)$ are LTL formulas. It is assumed that bound variable symbol b is not contained in V.

[1]We do not need to consider the weak until operator $\mathbf{W}$, and the release operator $\mathbf{R}$.

C. Bounded Trace Semantics for LTL

The semantic rules for evaluating LTL formulas on finite trace segments $s_i.s_{i+1} \ldots s_k$ are specified using notation $\langle\varphi\rangle_i^k$. The recursive rules for evaluating the truth value of $\langle\varphi\rangle_i^k$ can be directly transformed into an algorithm unrolling $\langle\varphi\rangle_i^k$ into a proposition no longer involving any temporal operators ($\mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}$), but referring to variable valuations in states $s_i, s_{i+1} \ldots, s_k$ and Boolean operators $\neg, \wedge, \vee$ only. Observe that we omit the semantics for $\mathbf{G}$ here, because our witnesses violating data rules are always represented by finite trace segments $s_i.s_{i+1} \ldots s_k$ without loops, whereas $\mathbf{G}\varphi$ only holds true if the trace segment has a lasso shape, where previous state on the segment is re-visited, thereby creating a cycle. The BMC semantics of $\mathbf{G}$ is discussed in detail in [20], [21].

The remaining transformation rules applicable for data validation are (symbols p denote atomic propositions)

$\langle\varphi\rangle_i^k = \text{false iff } i > k$
$\langle p\rangle_i^k$ iff $p[s_i(v)/v \mid v \in free(p)]$ Note that bound(p) $= \varnothing$
$\langle\neg\varphi\rangle_i^k$ iff $\langle\varphi\rangle_i^k$ is false
$\langle\varphi \wedge \psi\rangle_i^k$ iff $\langle\varphi\rangle_i^k$ and $\langle\psi\rangle_i^k$ are true
$\langle\varphi \vee \psi\rangle_i^k$ iff $\langle\varphi\rangle_i^k$ or $\langle\psi\rangle_i^k$ are true
$\langle(\exists b : \varphi)\rangle_i^k \equiv \langle\varphi\rangle_i^k \wedge \bigwedge_{j=i}^{k-1} (s_j(b) = s_{j+1}(b))$
Note that b occurs free in RHS formula
and extends domain of $s_j, s_{j+1}, \ldots, s_k$ by b
$\langle\varphi \mathbf{U} \psi\rangle_i^k \equiv \langle\psi\rangle_i^k \vee (\langle\varphi\rangle_i^k \wedge \langle\varphi[b'/b \mid b \in \text{bound}(\varphi)]\mathbf{U}\psi\rangle_{i+1}^k)$
$\langle\mathbf{X}\varphi\rangle_i^k \equiv \langle\varphi\rangle_{i+1}^k$
$\langle\mathbf{F}\varphi\rangle_i^k \equiv \bigvee_{j=i}^k \langle\varphi\rangle_j^k$

a) Example: Consider the BMC evaluation of property $(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)$ on trace segment $s_0.s_1.s_2$, that is $\langle(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)\rangle_0^2$. Applying the rules above, this is unrolled to

$\langle(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)\rangle_0^2 \equiv$
$\langle(x > 10)\rangle_0^2 \vee$
$(\langle(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\rangle_0^2 \wedge$
$\langle(\exists b' : y = b' \wedge \mathbf{X}(y = b' + 1))\mathbf{U}(x > 10)\rangle_1^2) \equiv$
$(s_0(x) > 10) \vee$
$(\langle(y = b) \wedge \mathbf{X}(y = b + 1))\rangle_0^2 \wedge$
$\bigwedge_{j=0}^1 (s_j(b) = s_{j+1}(b)) \wedge$
$\langle(\exists b' : y = b' \wedge \mathbf{X}(y = b' + 1))\mathbf{U}(x > 10)\rangle_1^2) \equiv$
$(s_0(x) > 10) \vee$
$((s_0(y) = s_0(b)) \wedge (s_1(y) = s_1(b) + 1)) \wedge$
$\bigwedge_{j=0}^1 (s_j(b) = s_{j+1}(b)) \wedge$
$((s_1(x) > 10) \vee$
$(s_1(y) = s_1(b') \wedge s_2(y) = s_2(b') + 1) \wedge (s_1(b') = s_2(b')) \wedge$
$\langle(\exists b'' : y = b'' \wedge \mathbf{X}(y = b'' + 1))\mathbf{U}(x > 10)\rangle_2^2) \equiv$
$(s_0(x) > 10) \vee$
$((s_0(y) = s_0(b)) \wedge (s_1(y) = s_1(b) + 1)) \wedge$
$\bigwedge_{j=0}^1 (s_j(b) = s_{j+1}(b)) \wedge$
$((s_1(x) > 10) \vee$
$((s_1(y) = s_1(b')) \wedge (s_2(y) = s_2(b') + 1) \wedge (s_1(b') = s_2(b')) \wedge$
$((s_2(x) > 10) \vee ((s_2(y) = s_2(b'')) \wedge \text{false})))$

D. Applications

We will now describe several examples illustrating the expressiveness of LTL for the verification of data validation rules.

b) Example: The simplest validation rules state that instances representing elements of a certain type $t = \tau$ must have certain attributes with values in a specific range, such as $a_i \in [x_0, x_1]$. A violation of this property is readily expressed by LTL formula $\mathbf{F}(t = \tau \wedge (a_i < x_0 \vee x_1 < a_i))$.

c) Example: The following rule checks the correctness of channel connections. *"If there exists a channel from s_1 to s_2, there must exist a channel in the reversed direction"*. A violation of this rule can be specified in natural language as *"There exists an instance s_1 which is not the auxiliary initial state, so that s_1 is connected to some instance s_2, but all channels emanating from s_2 lead to instances different from s_1"*. In LTL this is expressed as

$$\mathbf{F}(\exists i : id = i \wedge id > 0 \wedge \mathbf{X}(a \neq i \wedge b \neq i \wedge \ \ldots \ \wedge h \neq i))$$

A witness for such a rule violation reaches an element s with positive id (so it does not equal s_0) and at least one of its reachable neighbours (which, by definition of R, are only reachable if there is a connecting channel from s to this neighbour) has no channel with destination s.

d) Example: The following rule pattern frequently occurs when checking configuration data with respect to software component instances representing illegal sequences of track elements along a route. *"Following a track element of type τ_1 along its a-channel, and only regarding primary channel connections, an element of type τ_2 must occur, before an element of type τ_3 is found"*. The violation of this rule is specified by *"Find a track element of type τ_1 and follow it along its a-channel, so that only elements of type $t \neq \tau_2$ may be found along its primary channel directions, until an element of type τ_3 is encountered"*.

$$\begin{aligned}\mathbf{F}(t = \tau_1 \wedge \exists x : (a = x \wedge \mathbf{X}(id = x \wedge ((t \neq \tau_2 \wedge \\ \exists y : ((a = y \vee b = y \vee c = y \vee d = y) \wedge \mathbf{X}(id = y))) \\ \mathbf{U}(t = \tau_3)))))\end{aligned}$$

III. Verification of System Safety

This section describes our method for formally verifying safety of an interlocking system.

A. Formalization of the Verification Task

According to our method, the input of this verification step should consist of:

- a formal, state-based, behavioural model $\mathcal{M}$ of the interlocking system and its physical environment and
- safety conditions Φ expressed as a conjunction of propositions over the state variables in $\mathcal{M}$.

The verification goal is then to verify that the safety conditions Φ hold for any reachable state in $\mathcal{M}$.

As will be explained below, a model checker tool should be used for automated verification of such a goal. Therefore, the model $\mathcal{M}$ and the formula Φ should be expressed in the input language of the chosen model checker.

B. Verification Strategy

There is an established approach to apply bounded model checking in combination with inductive reasoning, in order to prove global system properties; this approach is called *k-induction*. For proving that safety condition Φ holds for all reachable states of $\mathcal{M}$, this method proceeds as follows.

1) First prove that $\Phi \wedge \Psi$ holds for the $k > 0$ first execution cycles after initialisation, i.e. $\Phi \wedge \Psi$ holds for $k > 0$ successive[2] states $\sigma_0, \ldots, \sigma_{k-1}$ of which σ_0 is the initial state of $\mathcal{M}$.
2) Next prove the following for an arbitrary execution sequence of $k+1$ successive states $\sigma_t, \ldots, \sigma_{t+k}$ of which the first σ_t is an arbitrary state (reachable or not from the initial state σ_0): if $\Phi \wedge \Psi$ holds in the k first states $\sigma_t, \ldots, \sigma_{t+k-1}$, then $\Phi \wedge \Psi$ must also hold for the $k+1^{st}$ state σ_{t+k}.

Here Ψ is an auxiliary property that holds for reachable states. (Note that Ψ is simultaneously proven by the given induction principle.) The proofs of the base case and the induction step should be performed by a bounded model checker tool. An example of such a tool is described in [23]. This tool treats the two proof obligations by exploring corresponding propositional satisfiable problems and solving these by a SAT solver. Note that the induction steps argue over an execution sequence of k+1 states of which the first state, σ_t, may be unreachable, although it would have been sufficient only to consider sequences for which σ_t is reachable. For sequences starting at an unreachable state, the induction step may fail and the property checker produces a *false negative*. To avoid this, the desired property Φ is strengthened with auxiliary property Ψ that is false for those unreachable states, σ_t, for which the induction step would otherwise fail.

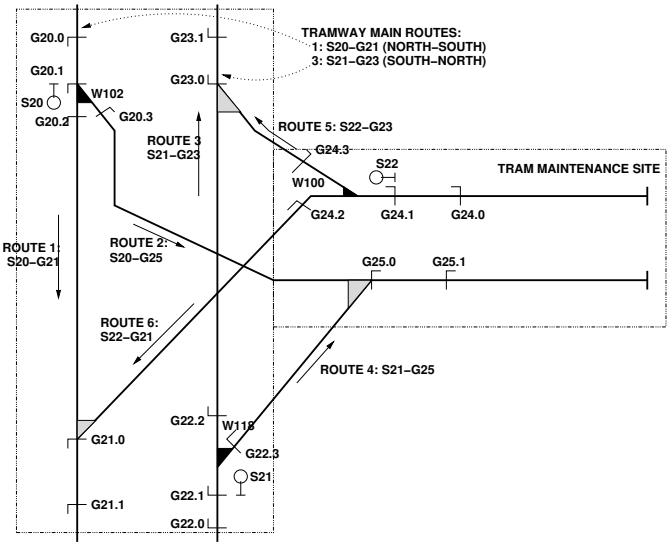


Fig. 2. A tramway network.

[2]Two states σ_i and σ_{i+1} are successive, if there is a transition from σ_i to σ_{i+1} according to $\mathcal{M}$.

C. Case Study

A reference publication for this verification technique has been published in [23]. It describes a real-world route-related tramway control system. For the network in Figure 2, the model of the tramway control system was verified to be safe, using k-induction. The safety conditions Φ was a conjunction of 15 conditions ensuring no collisions and no derailments of trams, and the auxiliary condition Ψ was a conjunction of conditions expressing state relations needed as assumptions in the induction step, in order to rule out unreachable states that would have given rise to false negatives otherwise. It turned out that a value of $k = 3$ sufficed to carry out the induction. The proofs of the base case and the induction step were performed by a bounded model checker, which used 392 seconds to perform the proofs. For more details about the case study, see e.g. [23], [3].

IV. Conclusion

In this article the application of bounded model checking for verification and validation of interlocking systems has been described. In contrast to global model checking which usually leads to state space explosions when applied to complex interlocking systems, bounded model checking allows for application in large and complex interlocking system layouts. It has been shown how the technique can be applied on two levels. First, in the form of LTL property checking, for the purpose of configuration data validation. Next, in combination with inductive reasoning, for the purpose of verifying safety properties for the dynamic behaviour of trains traversing the track network. Tool applications and measurements show that both application scenarios scale up for application in an industrial context.

Acknowledgments: The first author has been supported by the RobustRailS project funded by the Danish Council for Strategic Research. The second and third authors have been supported by the openETCS project funded by the European ITEA2 organisation.

References

[1] "Guide to the software engineering body of knowledge." [Online]. Available: http://www.computer.org/portal/web/swebok/home

[2] J. Pachl, *Railway Operation and Control*. VTD Rail Publishing, January 2002.

[3] A. E. Haxthausen, J. Peleska, and S. Kinder, "A Formal Approach for the Construction and Verification of Railway Control Systems," *Formal Aspects of Computing*, vol. 23, no. 2, pp. 191–219, 2011, the article is also available electronically on SpringerLink: http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s00165-009-0143-6.

[4] European Committee for Electrotechnical Standardization, *EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. Brussels: CENELEC, 2011.

[5] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi, "Model Checking Interlocking Control Tables," in *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2010)), Braunschweig, Germany*, E. Schnieder and G. Tarnai, Eds. Springer, 2011.

[6] A. E. Haxthausen and J. Peleska, "Efficient Development and Verification of Safe Railway Control Software," in *Railways: Types, Design and Safety Issues*. Nova Science Publishers, Inc., 2013, pp. 127–148.

[7] A. E. Haxthausen, "Towards a Framework for Modelling and Verification of Relay Interlocking Systems," in *16th Monterey Workshop: Modelling, Development and Verification of Adaptive Systems: the Grand Challenge for Robust Software*, ser. Lecture Notes in Computer Science, no. 6662. Springer, 2011, pp. 176–192.

[8] ——, "Automated Generation of Safety Requirements from Railway Interlocking Tables," in *5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'2012), Part II*, ser. Lecture Notes in Computer Science, no. 7610. Springer, 2012, pp. 261–275.

[9] D. Bjørner, "New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems," in *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003), Budapest/Hungary*. L'Harmattan Hongrie, May 15-16 2003.

[10] A. Fantechi, W. Fokkink, and A. Morzenti, "Some Trends in Formal Methods Applications to Railway Signaling," in *Formal Methods for Industrial Critical Systems*. John Wiley & Sons, Inc., 2012, pp. 61–84. [Online]. Available: http://dx.doi.org/10.1002/9781118459898.ch4

[11] K. Winter, "Optimising ordering strategies for symbolic model checking of railway interlockings," in *5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'2012), Part II*, ser. Lecture Notes in Computer Science, no. 7610. Springer, 2012, pp. 246–260.

[12] A. Fantechi, "Distributing the Challenge of Model Checking Interlocking Control Tables," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2012, vol. 7610, pp. 276–289. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34032-1_26

[13] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Defining and Model Checking Abstractions of Complex Railway Models using CSP||B," in *The 8th Haifa Verification Conference, November, 2012*, November to appear.

[14] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, "Météor: A successful application of b in a large project," in *FM'99 – Formal Methods*, ser. Lecture Notes in Computer Science, J. Wing, J. Woodcock, and J. Davies, Eds., vol. 1708. Berlin Heidelberg: Springer, 1999, pp. 369–387.

[15] M. Clabaut, C. Metayer, and E. Morand, "4B-2 formal data validation – formal techniques applied to verification of data properties," in *Embedded Real Time Software and Systems ERTS*, 2010. [Online]. Available: http://web1.see.asso.fr/erts2010/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0158_final.pdf

[16] T. Lecomte, L. Burdy, and M. Leuschel, "Formally checking large data sets in the railways," *CoRR*, vol. abs/1210.6815, 2012.

[17] A. E. Haxthausen and J. Peleska, "Formal Development and Verification of a Distributed Railway Control System," *IEEE Transaction on Software Engineering*, vol. 26, no. 8, pp. 687–701, 2000.

[18] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[19] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[20] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207. [Online]. Available: http://dl.acm.org/citation.cfm?id=646483.691738

[21] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded ltl model checking," *Logical Methods in Computer Science*, vol. 2, no. 5, pp. 1–64, 2006.

[22] A. P. Sistla, "Liveness and fairness in temporal logic," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 495–512, 1994.

[23] R. Drechsler and D. Große, "System level validation using formal techniques," *IEE Proc.-Comput. Digit. Tech.*, vol. 152, no. 3, pp. 393–406, May 2005.

Data Formal Validation of Railway Safety-Related Systems: Implementing the OVADO Tool

Robert Abo
Systerel
Les Portes de l'Arbois - Bâtiment A
1090 rue Descartes
13857 Aix-en-Provence Cedex 3
France
Email: robert.abo@systerel.fr

Laurent Voisin
Systerel
Les Portes de l'Arbois - Bâtiment A
1090 rue Descartes
13857 Aix-en-Provence Cedex 3
France
Email: laurent.voisin@systerel.fr

***Abstract*—This paper describes the process of data validation for railway safety-critical computer-based systems implemented by Systerel. More precisely, it describes the validation of data against their requirements, but it does not address another important task performed as part of data validation which is that data used by the systems really corresponds to the used physical railway. Standards, especially CENELEC EN 50128, recommend the use of formal methods for designing such systems. We use the OVADO formal tool to perform data validation. For that, we model data requirements by using the specification language of the B method, namely the B language, before using OVADO that automatically checks that data meet requirements. This tool integrates two independent components that must give the same results when they are applied on the same data, according to the principle of redundancy. An example of data validation for a CBTC system is also given.**

I. Introduction

Saying that present-day railway systems, moreover as most industrial systems (objects of our everyday life, cars, planes, plants, nuclear power stations, weapons, etc.) implement computer-based components is obvious, almost a commonplace, as computer-science controls our modern industries. But it is not trite to note that some of them implement a safety function, so that their malfunction or failure may have dramatic consequences on them and their users. This is the reason why their development requires a lot of rigor and discipline, leading to its own branch of computing known as *safeware* [1]. Otherwise, each electrical system is characterized by a *Safety Integrity Level* (SIL) which, as defined in IEC 61508 standard [2], denotes the risks involved in the system application by using a scale of 1 (the lowest risks) to 4 (the highest). Failures of SIL 3 or SIL 4 systems, aka *safety-critical systems*, may cause human casualties, severe damages or loss of expensive equipments and, also, environmental harms.

In Europe, the development of railway systems is governed by the legislation in force in a specific country and by international CENELEC[1] standards, which define objectives in terms of safety and security and the methods to reach them. These standards, which are all variations of IEC 61508[2], are: EN 50126[3] (for the methods to implement to demonstrate RAMS[2] of applications), EN 50128[4] (dedicated to the safety of software components[3]), and EN 50129[5] (devoted to the safety of hardware components). They are completed by EN 50159 dedicated to safety-related communications in closed (part 1) and in open transmission systems (part 2) [6].

In this paper, we focus on *Communication-Based Train Control* (CBTC) systems i.e. railway signalling systems that use telecommunications between trains and trackside equipment[7]. They aim at safely managing trains on an entire line, while absorbing the passenger traffic in particular during peak hours. There may be some differences depending on the technologies implemented by the different suppliers. But, basically, as illustrated in Figure 1, a railway signalling system has a pyramidal structure made of five layers. In the first one, at the base of its structure, we find the trackside equipment (i.e. rails, balises, points, signals, etc.). Just above, the equipments in charge of detecting a train on the track, i.e. track circuits, mainly. The first of the higher layers is composed of *Interlocking* (IXL) which is in charge of safely establishing and monitoring the train routes without any risks of collisions, catching up, and other traffic conflicts, for example with cars at crossings. Interlocking is SIL 4. The layer above is the *Automatic Train Control* (ATC) whose purpose is running a train while protecting it from dangerous situations. For that, it is composed of *Automatic Train Protection* (ATP) in charge of supervizing the train speed (ATP is SIL 4), and *Automatic Train Operation* (ATO) which automatically drives the train. A *Maintenance Aid System* (MAS) complements this layer. Finally, the *Automatic Train Supervision* (ATS) heads up the whole structure, and allows operators to remotely control railway traffic on an entire line.

Besides, many safety-related systems as CBTCs are wholly or partially composed of generic software elements, which are adapted to a particular application by means of application algorithms and/or *configuration data* ([4], Section 8). These static data describe the geographical arrangement of equipments and capability of the rail infrastructure and, therefore, they never change, contrarily to *dynamic data* which describe the current state of equipment along the track. Each software

[1]The *European Committee for Electrotechnical Standardization*), which is responsible for European standardization in the area of electrical engineering.

[2]RAMS stands for *Reliability*, *Availability*, *Maintenability* and *Safety*.

[3]This standard defines the notion of *Software SIL* inherited from that of IEC 61508 with a first level, SSIL 0, which denotes a non-safety-related software component.

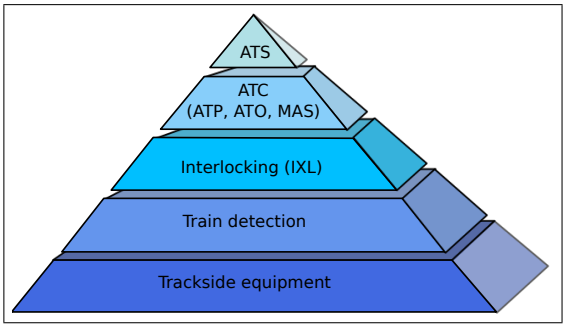


Fig. 1. The pyramidal structure of a railway signalling system

component of a CBTC, in particular the most critical ones, loads at runtime static data to perform its aim. This is why static configuration data plays a vital role in ensuring the safe operations of trains. Therefore, as hardware and software components, it needs to be validated. This is usually done in the earliest stages of a CBTC's life cycle by a dedicated team. In addition, for safety-critical software components including IXL and ATP, EN 50128 standard "highly recommends" the use of formal methods for their development process.

We describe here the management of data validation as it is done by Systerel. The core business of this medium-sized enterprise, located in Aix-en-Provence, Lyon, Toulouse and Paris in France, is the development of safety-critical embedded systems for rail transportation, aeronautics, energy, etc. It provides an expertise in the use of formal methods throughout the development cycle of a system, i.e. its design, its development and its validation[4].

This paper is organized as follows. Section I introduces data validation in the railway sector. Section II outlines its basics. Section III is entirely dedicated to the OVADO data validation tool used by Systerel's to conduct its projects. Section IV presents a real example of data validation of a CBTC. Finally, Section V concludes this article.

II. Background

In introduction, we have highlighted the need of validating data for CBTCs. But, what does it exactly mean? In this section, we define data validation before describing a process that implements it independently of any CBTC.

A. Definition

When we validate hardware and software components, we check that they meet their requirements by testing them or by proving it using formal methods. Similarly, we define data validation as *the process consisting in ensuring that data used by a safety-critical computer-based system conforms to a collection of requirements which define its usefulness, correctness and completeness for this system*. In other words, data validation consists in checking that data meet the requirements defined for this system.

[4]For further information, please visit http://www.systerel.fr

B. A semiautomatic process based on the B method

In the railway sector, data validation has been done entirely manually for a long time, leading to a tricky, fastidious, error-prone and long-term activity. For example, the authors note in [8], that it took more than six months to check that one hundred thousand data items were in accordance with two hundred properties representing requirements. Great R&D efforts have induced the design of industrial practical tools that now allow the semiautomatisation of data validation. This process is not fully automatic because, while the validation is automatically performed by a tool, the requirements still need to be manually modeled and these models need to be checked by dedicated engineers. Nevertheless, semiautomatisation has undoutedly increased the speed and level of confidence of data validation [8]. As mentioned in Section IV-D, let us note now that the way of modelling requirements has got a great influence on the performance of the tool.

On the other hand, known as a success story in software engineering for the railway sector, the *B method* is a formal method designed in the nineties by Jean-Raymond Abrial [9]. Let us briefly recall that software development with B consists in successively refining the models of a specification until obtaining an implementation which is automatically translated into source code. Each refinement step consists in introducing details in abstract models, and then in proving the consistency and compliancy of refined models with the abstract ones they refine. More recently designed, *Event-B* enlarges the scope of the B method with the purpose of studying and specifying whole systems, not only its software components [10]. We use a subset of the specification language of the B method, namely the *B language*[5] to model data requirements specified in natural language. The choice of the B language is explained by the implementation of B in software engineering for railways, its ease of use (although models are not always easy to write), and also its ease of learning. Let us note that a data validation process relying on Event-B has been also developped [11]. The B method and OVADO were chosen because they are part of the expertise and know-how of Systerel.Other formal languages and tools exist: for example, let us quote SCADE based on the synchronous language Lustre[6]. This is another approach of formal data validation which is not described in this paper. Ontologies for railway systems constitute a promising R&D axis [12].

C. An iterative process

The principle of data validation is illustrated in Figure 2. Firstly, high-level data requirements are modelled as predicates in set-theory. These predicates are expressed using the B language. Secondly, a tool automatically evaluates their truthfulness (true or false). In these conditions, data falls into two categories: on the one hand, the correct data that meet the requirements and for which predicates are true, and on the other hand the incorrect data that do not meet them and for which predicates are false.

For the latter, an analysis is performed to determine the origin of the non-conformity before restarting the process from

[5]A useful summary of the syntax of the B language can be found at http://www.stups.uni-duesseldorf.de/ProB/index.php5/Summary_of_B_Syntax.

[6]Further information about Lustre or model checking is available at http://www-verimag.imag.fr/Synchrone,30?lang=en

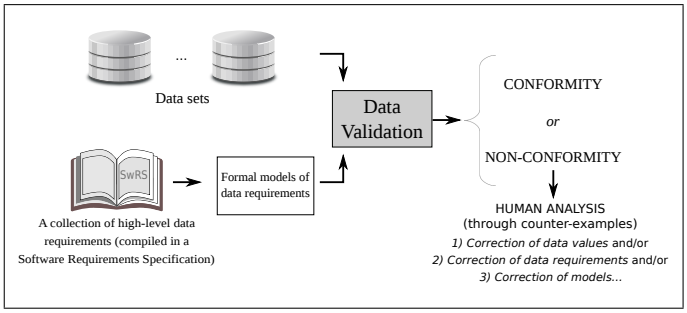


Fig. 2. Principle of data validation

its beginning. In effect, this is an iterative process in the sense that it is repeated after corrections are made until all data are compliant with the requirements. The origin of an error may be: the value of data itself; the requirements in natural language that have not been updated; or the model! Because, they can be wrong too and wrong models may validate faulty data!

Indeed, data validation is a great source of errors[7] especially when requirements are difficult to model[8]. We must highlight wrong data by really ensuring that data marked as correct is really correct. This is why the models of requirements need themselves to be validated prior to being able to validate data. Validating the models of data requirement aims at ensuring that wrong data are detected.

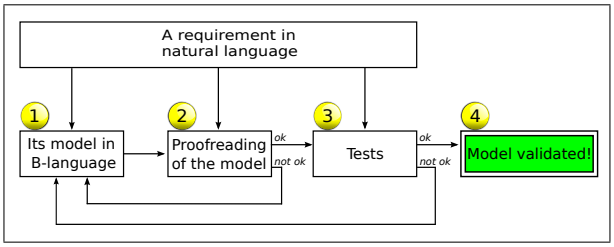


Fig. 3. Validation of the models

Therefore, the principle of data validation as illustrated in Figure 2 has to be refined. Modelling requirements is a four-stage process as illustrated in Figure 3. Stage 1 corresponds to the design of a model from the specification of a requirement in natural language (English, French, etc.). Several properties can be provided to cover a single requirement of the informal specification, with the purpose of simplifying the model. In order to reduce the risk of errors previously mentioned, each model is proofread by a reviewer who is different from the specifier who has written it, as required by EN 50128 Section 5 and 6 (stage 2). Then, the proofread model is tested by using a set of deliberately wrong data (stage 3). Again, as required by EN 50128, the tester is different from the designer who wrote the model. Moreover, he (or she) can only access the specification in natural language of the requirement, not the model under test (black-box testing). If an error is discovered, the model is corrected and the whole validation cycle, from the beginning, restarts (return to stage 2). Finally, when no error is found anymore, the model is approved and thus can be used to validate data (stage 4).

Proofreading consists in tracking down what we could call "over-specification" errors (i.e. the model specifies more things than the informal specification), and "under-specification" errors (i.e. the model specifies less things than the informal specification). These kinds of errors are usually due to a lack of understanding of the requirements, repetitions, oversights, non-updated models after a new release of the informal specification, sometimes minor, and so on. Let us add that the designer is not compelled to follow the proofreader's comments by justifying his (her) choice.

Verifying the models aims at checking that they translate well into the B language a requirement in natural language, and that the specifier does not make any foolish mistakes. But, tests are also performed to track tautologies i.e. predicates which are always true, whatever data is. For example, $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ being some predicates, $((\mathtt{A} \wedge \mathtt{B}) \Rightarrow \mathtt{C}) \Leftrightarrow (\mathtt{A} \Rightarrow (\mathtt{B} \Rightarrow \mathtt{C}))$ is a tautology. This kind of error is difficult to highlight by proofreading, this is the reason why wrong data is deliberately designed in order to render each property false at least once. If it is impossible that a property becomes false, then this property is a tautology and it must be corrected.

When completed, stages 2 to 4 produce some deliverables: the *proofreading report*, which summarizes the proofreads of all models; the *test report*, which aims at doing the same thing for the tests of models; and, of course, the *documentation of the models* which summarizes the models with their description and justification in natural language, and also a traceability matrix in order to show that all the informal requirements are covered by at least one property written in the B language.

Finally, to increase confidence in the results, data validation is done by using two independent tools that must draw the same conclusion of data compliancy when they are used to validate the same data. For data validation done at Systerel, these tools, *PredicateB* and *ProB*, are integrated in the OVADO platform which is the subject of the next section. To finish this section, we would like to quote that other data validation tool exist, such as Alstom's *DTVT* not presented in this paper.

III. The OVADO Tool

A. Overview

The RATP[9] initiated the development of OVADO[10], a formal tool in order to validate static data of the Paris's metro line 13 that was being automated.

This tool parses datasets (XML, Excel, text-based, or binary formats), loads properties and checks compliancy of data with respect to the loaded properties. The development of OVADO is now subcontracted to Systerel. It is composed of two different tools that form the basis of two independent data validation workflows:

[7] *Cuiusvis hominis est errare, nullius nisi insipientis in errore perseverare* i.e. "Any man can make mistakes: nobody but a fool will persist in error (Cicero, Philippicae XII, ii, 5)

[8] Let us quote an encountered real example of an indivisible requirement described in nineteen pages of a document. Its model has five hundred lines of predicates written in the B language.

[9] The *Régie Autonome des Transports Parisiens* is the firm in charge of the public transports in Paris, France.

[10] This acronym stands for *Outil de VAlidation de DOnnées* which means "Data Validation Tool" in French.

- The *PredicateB* predicate evaluator is in charge of checking the truthfulness of predicates modeling data requirements in the B language as explained in Section II); and
- ProB[11][13], is an animator and model checker for the B Method. It can be used to check a specification for range of errors. The constraint-solving capabilities of ProB can also be used for model finding, deadlock checking and test-case generation. ProB is currently developed by Michael Leuschel's team at the University of Düsseldorf in Germany, while its commercial support is provided by Formal Mind. Data and the models of requirements are converted into B models that are fed to the tool to validate data. This tool has been used with success on several projects (Roissy-Charles de Gaulle airport shuttle, Paris line 1, Barcelona line 9, Algiers line 1, etc.).

Data validation with OVADO is organized as follows. Data and formal properties of data requirements are fed into the tool. Properties are modelled as predicates in the B language. The conformance of the input data with the input requirements is independently validated by the two validation workflows of the tool. Each of them produces a validation report for each analysed property. The results of two reports relating to the same property must be equal. If not, the model or one of the tool is likely wrong and must be corrected. Let us add that OVADO has been applied with suceess to data validation of Paris lines 1, 3, 5, 13, and also Lyon's line B, etc.

B. Architecture

As shown in Figure 4, OVADO is a generic platform combining PredicateB and ProB, and can be completed by specific project plugins, such as the adaptation of OVADO for the acquisition of data described in a customer-specific format. Thus OVADO is able to be tailored to specific projects of industrials. OVADO can be used on a computer equiped with Microsoft Windows or Linux, and Java 6.

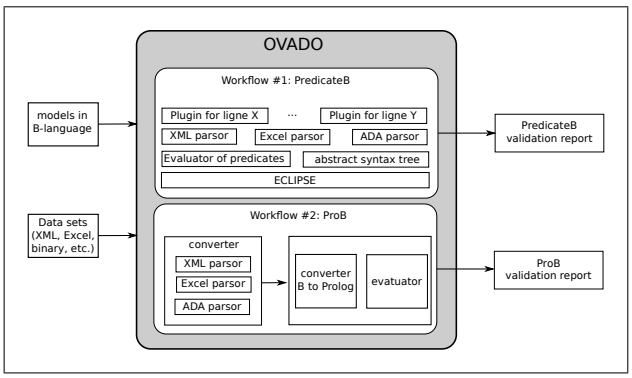


Fig. 4. Architecture of OVADO

C. User interface

The user interface of OVADO is illustrated in Figure 5 that shows the three main parts needed to write the models, perform data validation and analyse the results. Their organization

[11]The ProB website is http://www.stups.uni-duesseldorf.de/ProB/

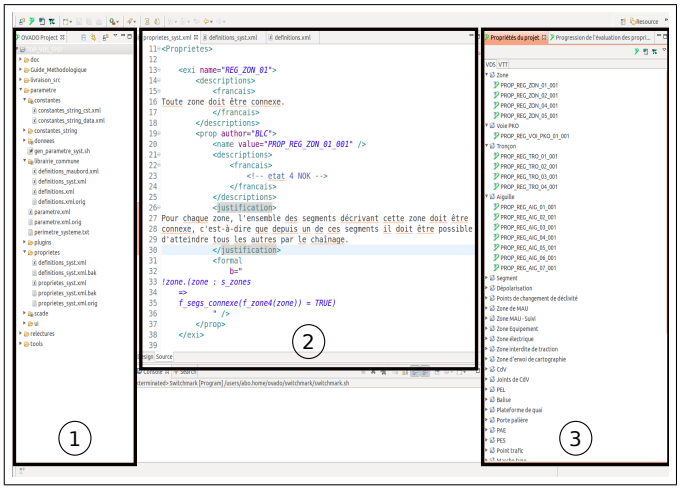


Fig. 5. The user interface of OVADO

within Eclipse depends entirely on how an engineer wants to organize his workspace. Part 1 displays the project tree. Part 2 shows different tabs where a designer can specify definitions and properties as explained in Section IV-B. Finally, Part 3 shows the "project properties" tab that lists all the specific properties of a system. By right-clicking on one of them, it is possible to launch its validation on the fly. A tab named "evaluation progress" shows the progress of the evaluation of a property. An example of the implementation of OVADO for validating data of a CBTC is detailled in the next section.

IV. Implementing OVADO

This section describes a project conducted by Systerel to validate the static data of a metro line. All names have been changed to respect the confidentiality of information.

A. Data

Datasets and models are structured in XML (*eXtensible Markup Language*) files. Each requirement is modeled by one or more properties. The decomposition of requirements in one or more properties is left to the discretion of a designer.

B. Models

1) Interface constants: The first step of the modelling activities consists in interfacing data with B constants used in the models. The following types can be defined to link the elements of datasets with constants of the properties written in the B language:

- carrier sets and their subsets;
- scalar data (mostly integers and character strings);
- relations between a scalar type and another scalar type;
- functions from a scalar type to another scalar type; and
- functions from a scalar type to functions of the previous type.

Each of them has a name, a predicate that specifies it (except for carrier sets) and a value, or a set of values, which is

the result of a XPath request, or several XPath requests in case of relations and functions, applied on the XML file defining real data as presented in Section IV-A.

Carrier sets define the different objects of the datasets used by a CBTC corresponding to the trackside equipment, the train detection equipment but also the organisation of the line according to a one-dimension Cartesian coordinate system, such as balises, signals, points, track circuits, blocks, etc. A block is an elementary portion of a railroad track, which has two extremities. Its origin extremity is at abscissa 0, while the abscissa of its destination extremity corresponds to its length. We also define four types of *zones* i.e. collections of blocks corresponding to four carrier sets: oriented (t_zoneori), non-oriented (t_zonenori), with two oriented extremities (t_zone2extrori) and, finally, with two non-oriented extremities (t_zone2extr). Oriented zones have *singularities* i.e. oriented extremities, while non-oriented zones have only extremities.

Scalar sets define constant numbers of the system such as abscissae, distances, speeds, temporary speed limits, delays, etc.

Relations and *functions* specify links between these objects such as the length of a block, the block where a particular object is located, its abscissa in millimeters on the block (an integer), the block which follows the current one in a particular direction, etc.

2) Definitions: In order to simplify the expression of properties, several libraries of definitions are defined prior to the modelling of properties themselves. Please note that these definitions are quite different from the macros one can define in a DEFINITIONS clause of a classical B machine, where syntax and semantics of expressions may lack of rigor. In the B method, our definitions would rather be declared as ABSTRACT CONSTANTS. A definition has a unique name, a description in natural language and an expression in the B language. They ease writing and understanding of properties, and checking them when proofreading.

For the project under consideration, a library of useful definitions was used to model graph functions. In effect, a railway network is represented by an oriented graph. Thus, the following r_zone2extr_extr relation gives the extremities of a zone with two extremities:

$$
\begin{array}{l}
\mathtt{dom}(\{\mathtt{zone}, \mathtt{extr}, \mathtt{numabsextr} \mid \\
\quad \mathtt{zone} \mapsto \mathtt{numabsextr} : \\
\quad \mathtt{t\_zone2extr} \ \lhd \\
\quad (\mathtt{f\_TabZones2Extr\_NumAbsExtr1} \\
\quad\quad \cup \ \mathtt{f\_TabZones2Extr\_NumAbsExtr2}) \ \land \\
\quad \mathtt{extr} \mapsto \mathtt{numabsextr} : \\
\quad\quad \mathtt{t\_extremite} \lhd \mathtt{f\_TabExtremities\_NumAbsolu}\})
\end{array}
$$

And, for each extremity, the f_extr_segdirabs function gives the triplet formed by its block, its direction and its abscissa on its block:

$$
\begin{array}{l}
\mathtt{ran}(\{\mathtt{extr}, \mathtt{seg}, \mathtt{dir}, \mathtt{abs}, \mathtt{res} \mid \\
\mathtt{xtr} \mapsto \mathtt{seg} : \mathtt{f\_extr\_seg} \ \land \\
\mathtt{extr} \mapsto \mathtt{dir} : \mathtt{f\_extr\_dir} \ \land \\
\mathtt{extr} \mapsto \mathtt{abs} : \mathtt{f\_extr\_abs} \ \land \\
\mathtt{res} = \mathtt{extr} \mapsto (\mathtt{seg} \mapsto \mathtt{dir} \mapsto \mathtt{abs})\})
\end{array}
$$

In the same manner, the r_zone2extrori_singu relation gives the singularities of a zone with two oriented extremities:

$$
\begin{array}{l}
\mathtt{dom}(\{\mathtt{zone}, \mathtt{sing}, \mathtt{numabssing} \mid \\
\quad \mathtt{zone} \mapsto \mathtt{numabssing} : \\
\quad \mathtt{t\_zone2extrori} \ \lhd \\
\quad (\mathtt{f\_TabZones2ExtrOri\_NumAbsSing1} \\
\quad\quad \cup \ \mathtt{f\_TabZones2ExtrOri\_NumAbsSing2}) \ \land \\
\quad \mathtt{sing} \mapsto \mathtt{numabssing} : \\
\quad\quad \mathtt{t\_singularite} \lhd \mathtt{f\_TabSingu\_NumAbsolu}\})
\end{array}
$$

And, the f_singu_segdirabs function gives, for each singularity, the triplet formed by its block, its direction and its abscissa on its block:

$$
\begin{array}{l}
\mathtt{ran}(\{\mathtt{singu}, \mathtt{seg}, \mathtt{dir}, \mathtt{abs}, \mathtt{res} \mid \\
\quad \mathtt{singu} \mapsto \mathtt{seg} : \mathtt{f\_singu\_seg} \ \land \\
\quad \mathtt{singu} \mapsto \mathtt{dir} : \mathtt{f\_singu\_dir} \ \land \\
\quad \mathtt{singu} \mapsto \mathtt{abs} : \mathtt{f\_singu\_abs} \ \land \\
\quad \mathtt{res} = \mathtt{singu} \mapsto (\mathtt{seg} \mapsto \mathtt{dir} \mapsto \mathtt{abs})\})
\end{array}
$$

The r_zonenori_extr relation gives the extremities of a non-oriented zone:

$$
\begin{array}{l}
\mathtt{ran}(\{\mathtt{zone}, \mathtt{extr}, \mathtt{numabsextr}, \mathtt{ind}, \mathtt{res} \mid \\
\quad \mathtt{zone} \mapsto \mathtt{ind} \mapsto \mathtt{numabsextr} : \\
\quad (\mathtt{t\_zonenori} \ \times \ \mathtt{INTEGER}) \lhd \\
\quad \mathtt{f\_TabZonesNOri\_NumAbsExtremites} \ \land \\
\quad \mathtt{numabsextr} / = -1 \ \land \\
\quad \mathtt{extr} \mapsto \mathtt{numabsextr} : \\
\quad\quad \mathtt{t\_extremite} \\
\quad\quad \lhd \mathtt{f\_TabExtremities\_NumAbsolu} \ \land \\
\mathtt{res} = \mathtt{zone} \mapsto \mathtt{extr}\})
\end{array}
$$

The following r_zoneori_singu relation gives the singularities associated with a particular oriented zone:

$$
\begin{array}{l}
\mathtt{ran}(\{\mathtt{zone}, \mathtt{singu}, \mathtt{numabssingu}, \mathtt{index}, \mathtt{res} \mid \\
\quad \mathtt{zone} \mapsto \mathtt{index} \mapsto \mathtt{numabssingu} : \\
\quad (\mathtt{t\_zoneori} \ \times \ \mathtt{INTEGER}) \lhd \\
\quad\quad \mathtt{f\_TabZonesOri\_ListSingZone} \ \land \\
\quad \mathtt{numabssingu} / = -1 \ \land \\
\quad \mathtt{singu} \mapsto \mathtt{numabssingu} : \\
\quad\quad \mathtt{t\_singularite} \\
\quad\quad\quad \lhd \mathtt{f\_TabSingu\_NumAbsolu} \ \land \\
\quad \mathtt{res} = \mathtt{zone} \mapsto \mathtt{singu}\})
\end{array}
$$

3) Properties: Each property has a name, a tag recalling the requirement it refers to, a description in natural language and a formal description in the B language. The requirement tags are used for the sake of traceability, in order to ensure that all requirements that must be modelled have been effectively modelled. Models should not be too complex to be easily proofread. In particular, definitions should be intensively used.

Let us consider the informal requirement "each zone must be connected". It means that for each zone, all blocks describing it (a zone is a collection of blocks) should be connected meaning that from one of these blocks, it must be possible to reach any other block by connection. With the previous interface constants and definitions except the definition of the $f\_zone\_connexe$ which is not given in this paper, the model of this requirement is specified as follows:

TABLE I. RESULTS OF DATA VALIDATION WITH PREDICATEB (A COMPONENT OF OVADO)

Component (and its amount)	Number of properties	Number of lines in B	Validation time in minutes for a component (and total)
#1 (13)	39	2050	<1 (<10)
#2 (4)	28	1177	<1 (<4)
#3 (4)	369	19613	180 (720 i.e. 12h)
#4 (1)	62	2741	< 1 (idem)
#5 (34)	159	12400	15 (510 i.e. 8.5h)
#6 (1)	26	1641	6 (idem)

$$
\begin{array}{l}
\forall(\mathtt{typezone}, \mathtt{r\_zone\_extr}, \mathtt{zone}, \mathtt{extrs}).(\\
\quad \mathtt{typezone} : \mathtt{0..3} \quad \wedge \\
\quad \mathtt{r\_zone\_extr} = \\
\qquad \{\, \mathtt{0} \mapsto (\mathtt{r\_zone2extr\_extr}; \mathtt{f\_extr\_segdirabs}) \\
\qquad , \mathtt{1} \mapsto (\mathtt{r\_zone2extrori\_singu}; \mathtt{f\_singu\_segdirabs}) \\
\qquad , \mathtt{2} \mapsto (\mathtt{r\_zonenori\_extr}; \mathtt{f\_extr\_segdirabs}) \\
\qquad , \mathtt{3} \mapsto (\mathtt{r\_zoneori\_singu}; \mathtt{f\_singu\_segdirabs}) \\
\qquad \}(\mathtt{typezone}) \quad \wedge \\
\quad \mathtt{zone} : \mathtt{dom}(\mathtt{r\_zone\_extr}) \quad \wedge \\
\quad \mathtt{extrs} = \mathtt{r\_zone\_extr}[\{\mathtt{zone}\}] \\
\quad \Rightarrow \\
\quad \mathtt{f\_zone\_connexe}(\mathtt{extrs}) = \mathtt{TRUE})
\end{array}
$$

The automatic treatment performed by OVADO has learned us that initial data sets did not meet this requirement: four oriented zones were not connected, i.e. there was no communication channel between them. Data sets were therefore corrected by the teams in charge of the definition of data before being successfully validated, in particular against this requirement, by the validation team.

C. Final results

We have modelled the data requirements for six components of a CBTC both carborne and trackside. Table I summarizes the results obtained with PredicateB only. To preserve confidentiality, the component names have been changed.

D. Influence of models on performance

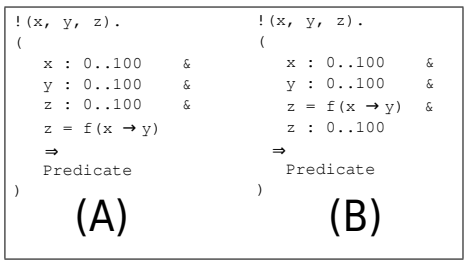


Fig. 6. A simple example that shows that models influence performance

Figure 6 shows two model patterns. In (A), OVADO checks the rule $\mathtt{z} = \mathtt{f}(\mathtt{x} \mapsto \mathtt{y}) \Rightarrow \mathtt{Predicate}$ for each triplet $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ verifying $\mathtt{x} \in [\mathtt{0..100}]$ and $\mathtt{y} \in [\mathtt{0..100}]$ and $\mathtt{z} \in [\mathtt{0..100}]$. That is to say that OVADO will create one million triplets then check this rule for each of them. In (B), OVADO checks the rule $\mathtt{z} : \mathtt{0..100} \Rightarrow \mathtt{Predicate}$ for each triplet $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ verifying $\mathtt{x} \in [\mathtt{0..100}]$ and $\mathtt{y} \in [\mathtt{0..100}]$ and $\mathtt{z} = \mathtt{f}(\mathtt{x} \mapsto \mathtt{y})$. That is to say that, OVADO will only create ten thousand triplets and will only perform ten thousand checks. This is also true for properties that use existantial quantifiers and/or sets defined by comprehension.

V. CONCLUSION

In this paper we have described the process for validating data used for safety-related railway systems. This process, which relies on the B method, presents several benefits: using formal methods is recommanded by international standards, the B language is quite easy to learn and to use, it is well suited for modelling requirements of CBTCs, large datasets can be used while the validation time is reasonable. On the contrary, let us face is it ill-suited for proving performance requirements and, unfortunatly, applied only in the railway industry at the moment.

ACKNOWLEDGEMENT

The authors would like to thank their teamates involved in data validation both based in Aix-en-Provence and Paris. This paper summarizes their work. Their gratitude is also adressed to Mr F. Bustany, President of Systerel, for allowing the writing of this paper.

REFERENCES

[1] N. G. Leveson, *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995.

[2] *IEC 61508 : Functional safety of electrical/electronic/ programmable electronic safety-related systems*, International Electrotechnical Commission Std.

[3] *EN 50126: Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, European Committee for Electrotechnical Standardization (CENELEC) Std.

[4] *EN 50128: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, European Committee for Electrotechnical Standardization (CENELEC) Std., 2011.

[5] *EN 50129: Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*, European Committee for Electrotechnical Standardization (CENELEC) Std.

[6] *EN 50159: Railway applications - Communication, signalling and processing systems - Safety-related communication in transmission systems*, European Committee for Electrotechnical Standardization (CENELEC) Std., 2010.

[7] *IEEE Std 1474.1-2004, IEEE Standard Method for CBTC Performance and Functional Requirements*, IEEE Std.

[8] T. Lecomte, L. Burdy, and M. Leuschel, "Formally checking large data sets in the railways," *CoRR*, vol. abs/1210.6815, 2012.

[9] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996, iSBN:0-521-49619-5.

[10] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[11] F. Badeau and M. Doche-Petit, "Formal data validation with event-b," *The Computing Research Repository (CoRR)*, vol. abs/1210.7039, 2012.

[12] M. Lodemann and N. Luttenberger, "Ontology-based railway infrastructure verification - planning benefits," in *KMIS*, K. Liu and J. Filipe, Eds. SciTePress, 2010, pp. 176–181.

[13] M. Leuschel and M. Butler, "Prob: A model checker for b," in *FME 2003: FORMAL METHODS, LNCS 2805*. Springer-Verlag, 2003, pp. 855–874.

Validation of Railway Interlocking Systems by Formal Verification, a Case Study

Andrea Bonacchi*, Alessandro Fantechi*, Stefano Bacherini†, Matteo Tempestini† and Leonardo Cipriani†
*Dipartimento di Ingegneria dell'Informazione
Università di Firenze, Florence, Italy
Email: a.bonacchi@unifi.it, fantechi@dsi.unifi.it
†General Electric Transportation Systems, Florence, Italy
Email: stefano.bacherini@ge.com, matteo.tempestini@ge.com, leonardo.cipriani@ge.com

***Abstract*—Notwithstanding the large amount of attempts to formally verify them, railway interlocking systems still represent a challenging problem for automatic verification. Interlocking systems controlling sufficiently large stations, due to their inherent complexity related to the high number of variables involved, are not readily amenable to automatic verification, typically incurring in state space explosion problems. The study described in this paper aims at evaluating and experimenting the industrial application of verification by model checking for this class of systems. The choices made at the beginning of the study, also on the basis of specific requirements from the industrial partner, are presented, together with the advancement status of the project and the plans for its completion.**

I. Introduction

In the railway signalling domain, an *interlocking* is the safety critical system that controls the movement of the trains in a station and between adjacent stations. The interlocking monitors the status of the objects in the railway yard and allows or denies the routing of the trains in accordance with the railway safety and operational regulations that are generic for the region or country where the interlocking is located. The instantiation of these rules on a station topology is stored in the part of the system named *control table*. Control tables of modern computerized interlockings are implemented by means of iteratively executed software controls over the status of the yard objects.

One of the most common way to describe the interlocking rules given by control tables is through boolean equations or, equivalently, ladder diagrams which are interpreted either by a PLC or by a proper evaluation engine over a standard processor. A first concern in the history of computerized interlockings has been the automatic generation of such boolean equation sets starting from generic signalling principles and from the topology of the layout of the station [1]. On the other hand, the certification activities for an interlocking include the verification that the implemented control tables actually satisfy safety rules. Verification of correctness of control tables has been a prolific domain for formal methods practitioners, and the literature counts the application of several techniques to the problem, namely the Vienna Development Method (VDM) [2], property proving [3], [4], Colored Petri Nets (CPN) [5] and model checking [6], [7]. This last technique in particular has raised the interest of many railway signalling industries, being the most lightweight from the process point of view, and being rather promising in terms of efficiency.

However, due to the high number of boolean variables involved, automatic verification of sufficiently large stations typically incurs in combinatorial state space explosion problem.

The first applications of model checking have therefore attacked portions of an interlocking system [8], [9]; but even recent works [10], [11] show that routine verification of interlocking designs for large stations is still out of reach for symbolic model checker NuSMV [12] and explicit model checker SPIN [13], although specific optimizations can help [11]. As we argument later, SAT-based model checking appear to be more promising at this respect.

We want however to notice that control tables may have two main roles (not always both present) in the development of these systems: either as specifications of the interlocking rules [14], often issued by a railway infrastructure company, or as implementations, when they come encoded in some (typically proprietary) executable language. Hence also verification may address different problems, such as the consistency of the former, or the correctness of the latter w.r.t. the former, or the check of safety properties on the latter. In the study presented in this paper, we address the last mentioned verification problem. Anyway, a typical issue of any of these verification tasks is the choice of how to express control tables in a language suitable for the verification tool adopted.

Indeed, commercial solutions exist for the production of interlocking software, such as Prover Technology's (Ilock), that includes formal proof of safety conditions as well, by means of a SAT solving engine. Industrial acceptance of such "black-box" solutions is however sometimes hindered by the fear of vendor lock-in phenomena and by the loss of control over the production process.

In the Safety and Validation Laboratory (S&V Lab) of General Electric Transportation System (GETS), with the final aim of reducing the costs of verifying the safety requirements of the produced interlocking systems, a feasibility study has been started, conducted in collaboration with the

PhD School of Information Engineering of the University of Florence, on the verification of legacy control tables that control a portion of a railway yard.
Indeed, the S&V Lab is acting (according to CENELEC EN50128 norms) as an independent verifier of the interlocking systems produced by other branches of the company, with little insight of the followed process, and focusing on the final product. Actually, the only information available on the implemented control tables can be extracted from the binary files, that are written in the target using a proprietary format, by means of libraries, that we will refer from now on as *legacy libraries*, provided by the interlocking developers.
In a previous exploratory work [10] the control tables were modelled as finite state machines and safety properties were proved by means of NuSMV. In this case, the choice of the tool and hence of the modelling language was taken instead according to specific constraints posed by the S&V Lab of GETS: in order to smoothly adopt this verification technique inside the internal production process, it was required that the verification tool is a commercial tool, already known within the company. Moreover, the difficulties encountered in dealing with medium and large size interlocking systems by means of BDD-based verification pointed to the alternative of adopting a SAT-based model checker, in order to exploit at best the native boolean coding coming from the control tables [15]. The conjunction of these constraints has favoured the choice of Matlab *Design Verifier*, which is based on a SAT solver, using boolean functions with logical gates as the language in which to translate the legacy control tables.
The commercial constraints posed by the company are due to a precise industrial policy, that is, minimizing additional investment, minimizing dependency from external suppliers, especially if not yet already known, while internally master the overall verification process.
This paper describes the current advancement of the feasibility study concentrating on the modelling phase. First we describe the ladder logic, that is, the industrial standard graphical language to represent boolean functions involved in control tables. In section 3 we introduce the modelling process and in particular the algorithm *LLD Parser* that allows the control tables to be translated into boolean functions that will be implemented in a Simulink model. On an example model some preliminary verification experiments have been carried out, showing that medium size interlockings can be verified using this approach (Section 4), and so confirming the initial intuition about using SAT-based verification tools.

II. Ladder Logic Diagrams

Ladder Logic is a graphical language which can represent a set of boolean equations of the type $x_i := e_i$, with $e_i = f(x_j, \cdots, x_{j+n})$, where f is a boolean function built as a composition of and,or, and not operators, and $x_i, x_j, \ldots, x_{j+n}$ are variables which represent the possible states of the signalling devices. Ladder Logic represents the working of relay-based control systems. For this reason the variables on the right expression of the equation are also named *contacts*, while the variables in the left hand are named *coils*. Variables can be distinguished in:

- *Input variables*: the value is assigned by sensor readings or operator commands. These variables are defined in the expressions e_i and cannot be used as coil.
- *Output variables*: can be only a coil and their value is determined by means of the assignments of the diagram and is delivered to actuators.
- *Latch variables*: the value is calculated by means of the assignments, but is used only for internal computation of the values of other variables. A latch variable is used as coil in an assignment and is an input variable in other assignments.

With these three kinds of variables, a *Ladder Logic Diagram* (LLD) describes a state machine whose memory is represented by the latch variables and the evolution is described by the assignment set. An execution of this state machine, named *control cycle*, involves:

1) Reading input variables; the values of these variables are assumed to be constant for the entire duration of the control cycle.
2) Computation of the current values for the output variables and for the latch variables starting from the values of the input variables and the values of the latch variables at the previous control cycle.
3) Transmission of the values of the output variables.

An example of a single row of a ladder logic diagram is reported in figure 1, expressing the boolean equation:

$$y = x \wedge (w \vee \neg z)$$

In this graphical language, if x is a boolean variable, an expression e can be defined in inductive way by means of following syntax:

- $--[\ ]--$ represents a variable.
- $--[/]--$ represents the negation of a variable.
- $--(\ )$ represents a coil.

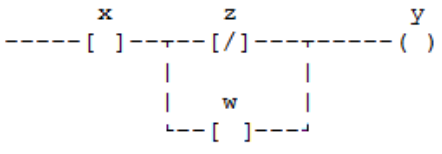


Figure 1. Example of ladder logic diagram

In general, a Ladder Logic Diagram expresses a set of boolean equations that can be written:

$$\tilde{x} = f(\tilde{x}, \tilde{y})$$

where $\tilde{x}, \tilde{y}$ are boolean variable vectors representing respectively state/output variables and input variables: these equations are cyclically executed. Let us call $\tilde{x}_i, \tilde{y}_i$ the vectors of values taken by such variables in successive executions. From the equations we can define $F(\tilde{x}_i, \tilde{x}_{i+1}, \tilde{y}_i)$ as a boolean function that is true iff $\tilde{x}_{i+1} = f(\tilde{x}_i, \tilde{y}_i)$, representing one execution of the equations. Let be $Init(\tilde{x})$ a predicate which is true for the initial vector value of state and output variables. If $P(\tilde{x})$ is a predicate telling that a desired (safety) property is verified by the vector $\tilde{x}$, then the following expression:

$$\Phi(k) = Init(\tilde{x}_0) \wedge \bigwedge_{i=0}^{k-1} F(\tilde{x}_i, \tilde{x}_{i+1}, \tilde{y}_i) \wedge \bigvee_{i=0}^{k} \sim P(\tilde{x}_i)$$

is a boolean formula that tells that P is not true for the state/output vector for some of the first k execution cycles. According to the Bounded Model Checking (BMC) principles [16], using a SAT-solver to find a satisfying assignment to the boolean variables ends up either in unsatisfiability, which means that the property is satisfied by the first k execution cycles, or in an assignment that can be used as a counterexample for P, in particular showing a k-long sequence of input vectors that cause the safety problem with P.

Due to the constraints we have discussed above on the choice of Design Verifier as a verification tool, the BMC working details are no more addressed in the following, since they are hidden in the tool itself. We need instead to focus on the representation in a format suitable for Design Verifier of the legacy control tables that are loaded, in the form of LLDs, in the analysed interlocking systems.

III. Model extraction

The first activity in the feasibility study has therefore addressed the definition of a process that allows a model of a station to be obtained from the analysed implementation in three steps:

1) **Import Station Data**: all data about a station (equations, timers, interfaces, ...) are imported in Matlab by means of the legacy libraries that read the binary files loaded on the interlocking system.
2) **Model Station Data**: the equations and the links between them are modelled in a Simulink model by means of the *LLD-Parser*.
3) **Model Properties**: safety properties are modelled with reference to the station model and are proved by means of Design Verifier.

A. Importing Data Station

As discussed in section II the boolean equations of an interlocking are represented in a ladder logic diagram (figure 1), which is encoded in a proprietary binary format for the diagram interpreter engine.

In order to extract this information from the binary code, we use those proprietary interpretation routines that we have called "legacy libraries". These libraries allow each boolean equation to be read as a matrix $M^{n \times k}$. The matrix is just a one to one representation of the ladder diagram with numeric codes. The code values in the matrix can be either positive, representing variables, or negative, representing either a connector or the polarity of a variable (see table I).

Table I
SYMBOL TRANSLATION

Symbol	Value	Symbol	Value
⌊	-1	− − [ ] − −	-10
⌋	-2	− − [/] − −	-20
⊥	-3	− − ( )	-30
⊤	-4	Blank space	-40
⊢	-5	Horizontal line	-50
⊣	-6	Vertical Line	-70
+	-7		

The LLD in figure 1 is for example encoded by the following matrix M:

$$\begin{bmatrix} -40 & 100 & -40 & 200 & -40 & 500 \\ -50 & -10 & -4 & -20 & -4 & -30 \\ -40 & -40 & -70 & 300 & -70 & -40 \\ -40 & -40 & -1 & -10 & -2 & -40 \end{bmatrix}$$

The values 100, 200, 300 and 500 are respectively associated to the variables x, z, w and y.

B. LLD Parser

The extracted matrix needs to be interpreted in order to define the boolean function it implements, expressed in a format suitable for Design Verifier. Three alternative ways to describe these functions are possible in Matlab, that is, using boolean gates in a Simulink diagram, using truth tables, or, by taking into account the typical cyclic execution of the equations as well, using a Stateflow state machine. Some preliminary experiments have suggested that the latter choice was employing the less direct correspondence with the boolean equations, and hence less prone to be efficiently handled by Design Verifier. We have for the moment chosen the first alternative, leaving a more accurate efficiency comparison as a future work.

We have hence designed an algorithm that translates the matrix into Simulink boolean and/or/not gates.

If we focus on the graphical format of LLD, we recognize one or more *connectors* which belong to the following set:

$$C = \{\lfloor \quad \rfloor \quad \perp \quad \top \quad \vdash \quad \dashv \quad +\}$$

Considering specific pairs of connectors, in the set C, it is possible to define a *connection relation* (CR) between them,

which defines a particular conjunction/disjunction between the variables in a LLD:

$$CR = \{(\lfloor, \rfloor), (\bot, \rfloor), (\lfloor, \bot)\}$$

The connection relation is the basis to provide semantics to a LLD. By means of this relation we can classify LLDs in a few *Families of Equations* (FoE).
A FoE is a set of Ladder Logic Diagrams that share some common graphical features. For example, the diagram in figure 2 represents the boolean equation:

$$y_1 = x_1 \wedge x_2 \wedge ((x_3 \wedge \neg x_4) \vee (x_5 \wedge \neg x_6 \wedge x_7) \vee (y_1 \wedge \neg x_8))$$

and belongs to the same FoE of the diagram in figure 1 because both equations have an *or gate* connected to an *and gate* but, for the second equation, the *or gate* has three inputs (three *and gate*); while the first equation has two variables in input.

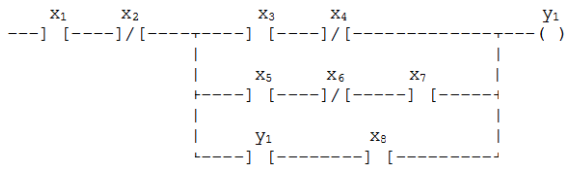


Figure 2. Example of ladder logic diagram

We need to find these patterns (pairs) to model the equations by means of logical gates. In fact, the LLD Parser (reported in algorithm 1) visits four times the matrix M; the first time it discovers all the input/output variables (positive values in the odd rows of M), then it reads the even rows of M, which contain the polarity of the variables (that is if they are asserted and/or negated in the equation).
Finally, if there is at least a logic *or*, that is, the value corresponding to the symbol $\top$ is present in the equation matrix M (*line* 3), the LLD Parser looks for the FoE to which the matrix M belongs and then runs a *Depth First Search* (DFS) on the connectors that are in the equation; otherwise this means that all the variables in the equation are in logic *and* (*line* 15).
The algorithm 2 is the DFS for the LLDs reported in figures 1 and 2 and the subrelations defined are: $CR_1 = \{\lfloor, \rfloor\}$, $CR_2 = \{\vdash, \dashv\}$ and $CR_3 = \{\top, \top\}$.

Sorting connectors (*lines* 2-3): the connectors in the matrix (C_M) are sorted from the deepest (greatest row and column in M) to the shallowest (C_{MO}).

Main Loop (*lines* 4-16): from the set C_{MO} a connectors pair (c_1, c_2) is extracted and if the pair belongs to a connection subrelation the variables are linked accordingly. In particular:

Create a new or gate (*lines* 5-10): if (c_1, c_2) belongs to CR_1 an *or* gate is created and the variables $var_{pattern}$ between the connectors (c_1, c_2) are connected, possibly through an *and* gate, to the new *or* gate; this construction is done by the function *LinkVariables*. At last, the variables in $var_{pattern}$ are deleted from the set var which contains all not yet connected variables.

Link other variables (*lines* 11-15): the case in which (c_1, c_2) belongs to CR_2 or CR_3 is similar to the previous cases, but no new *or* gate is built and the $var_{pattern}$ variables are connected to the most recently created *or* gate by the function *LinkVariables*.

Create final and gate (*lines* 17-19): if there are still variables in the set var (see the example of variable x for the LLD in figure 1 and variables x_1, x_2 in figure 2), they are linked with the most recently created *or* gate to a final *and* gate; otherwise the most recently created *or* gate is the final gate.

Algorithm 1 LLD Parser

Require: M equation matrix
Ensure: Model of the equation
1: $var \leftarrow$ GetVariables(M)
2: $syntax \leftarrow$ GetVariablePolarity(M,var)
3: **if** $\top \in M$ **then**
4: $family \leftarrow$ GetFoE(M)
5: **switch** $(family)$
6: **case** 1**:**
7: $DFS_1(M, var, syntax)$
8: **case** 2**:**
9: $DFS_2(M, var, syntax)$
$\vdots$
10: **case** N**:**
11: $DFS_N(M, var, syntax)$
12: **default:**
13: **end switch**
14: **else**
15: $LogicAnd(var)$
16: **end if**

After the parsing of the matrix M the output of the equation can: (1) activate a timer, (2) be input to the same equation. In the first case a timer is modelled and the output of the equation is linked to the timer, in the second case a delay block is created.
All the equations are then linked between them by means of the latch variables, or by timers when needed; in this way the model of a station is completed (see left part of Figure 3). The model has as input and output the input/output variables of the equations.

IV. Preliminary Results

The S&V Laboratory has carried out some preliminary experiments with the process and algorithm discussed in the previous section considering a network of four Computer Interlocking Subsystems: $(CIS_1, CIS_2, CIS_3, CIS_4)$ that

Algorithm 2 Depth First Search

$numOrGate \leftarrow 0$
$C_M \leftarrow C \in M$
$C_{MO} \leftarrow Order(C_M)$
for all $(c_1, c_2) \in C_{MO}$ **do**
  if $(c_1, c_2) \in CR_1$ **then**
    $numOrGate \leftarrow numOrGate + 1$
    $var_{pattern} \leftarrow var \in (c_1, c_2)$
    $LinkVariables(var_{pattern}, M)$
    $var \leftarrow DeleteVariables(var_{pattern}, var)$
  end if
  if $(c_1, c_2) \in CR_2 || (c_1, c_2) \in CR_3$ **then**
    $var_{pattern} \leftarrow var \in (c_1, c_2)$
    $LinkVariables(var_{pattern}, M)$
    $var \leftarrow DeleteVariables(var_{pattern}, var)$
  end if
end for
if $var \in var_{pattern}$ **then**
  $CreateAndFinal(var, numOrGate)$
end if

controls a small railway station.
The network has 2625 equations, 717 inputs and 915 outputs; each equation can have from one input to a maximum of 25 inputs. The number of equations, inputs and outputs, apportioned to single CISs is reported in table II.

Table II
DATA OF SINGLE CIS

CIS	Num. Equations	Num. Inputs	Num. Outputs
CIS_1	77	44	52
CIS_2	1608	370	522
CIS_3	430	151	157
CIS_4	511	152	184

An example of property P that has been defined and proved on CIS_3 is the following:

Under the preconditions:

1. The input from a track circuit A gives it as *unoccupied*.
2. A predefined time period has elapsed.
3. The input from the adjacent track circuit B, in accordance with the driving direction, is *occupied*.

the modelled state of A passes from *occupied* to *unoccupied*.

The property P is modelled in Simulink by the diagram in Fig 3. To prove the property P a small number of input/output variables was used (see figure 4) and Design Verifier has generated a counterexample; that is an input variables assignment that does not satisfy the property P. Each input variable in the counterexample assumes the

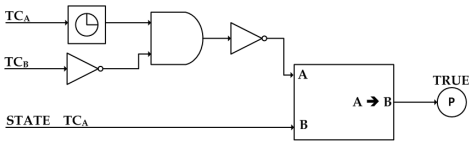


Figure 3. Property P

values: *true* (1), *false* (0) or *don't care* (-). Due to the high complexity of the interlocking logic, interpretation of the counterexample is not immediate, and requires the help of signalling engineers, who are able to distinguish real counterexamples from unfeasible combinations of inputs. At the current stage of the project, this particular activity has not yet been started. The first experiments were rather aimed at testing the capability of Design Verifier to deal with models of this size.
The entire process of importing data from the binaries, modelling the station and proving the property P (with a generation of a counterexample) has been run on an AMD Athlon(tm) II X2 B24 3GHz, 4GB of RAM machine with Windows 7, 32 bits, operating system. In table III, we report the times (in seconds) of the three phases.

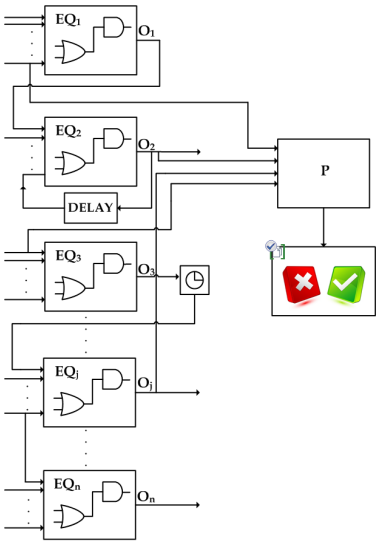


Figure 4. Prove Property

Table III
TIMES

Phase	Time (sec)
Import Data	60,334
Model Station	2506,271
Prove Property	2,000
Total	2568,605

V. Conclusion

In this paper we have reported a solution to model an interlocking system and to prove the correctness of safety properties P on the model.

We have implemented an algorithm that: (1) reads the data station by the binary files, that is loaded on the target, by means of legacy library; (2) parses the boolean equations, that are written in ladder logic, and generates a model which contains the equations and the station interfaces towards the adjacent stations.

To model the boolean equations we have defined the semantic of the ladder logic. The algorithm runs a depth first search in the ladder logic diagram to find out the connection patterns and by means of these patterns the algorithm builds the equation model. Finally all equation models are linked between them and the station model is created.

In the first experiments carried on, the algorithm has been applied on an interlocking system of 2625 equations and 717 input and 915 output interfaces that controls a portion of a railway station, obtaining the station modelling in less than one hour; finally the verification of a safety property has been attempted by means of Design Verifier, raising the problem of the interpretation of counterexamples obtained by verification, which may require the help of signalling engineers, to distinguish real counterexamples from unfeasible combinations of inputs. In order to rule out such unfeasible counterexamples, we plan to shape our verification process according to the CEGAR (CounterExample Guided Abstraction Refinement) paradigm [17], in order to provide an automated method adoptable in an industrial context.

The current experiments are focused on providing verification results on a set of production interlocking cases of different sizes. We will then address a deeper analysis of these results, focusing in particular on the optimization of the model to better exploit the underlying SAT solver of Design Verifier. At this regard, the possible alternative choices (state machine, truth tables) for modelling the control tables will be compared w.r.t. the verification performance of the tool. Moreover, we shail investigate the application of other verification tools, such as NuSMV, to the extracted data, in order to compare results and performance issues. This activity, although a change of the verification engine in the defined verification process is not planned by the company, will help to consolidate it, and will provide interesting compared data about the application of formal verification tools on industrial production case studies.

References

[1] B. Fringuelli, E. Lamma, P. Mello, and G. Santocchia, "Knowledge-Based Technology for Controlling Railway Stations," *IEEE Expert: Intelligent Systems and Their Applications*, pp. 45–52, 1992.

[2] K. M. Hansen, "Formalising Railway Interlocking Systems," in *Proceedings of the 2nd FMERail Workshop (1998)*.

[3] A. Borälv, "Case Study: Formal Verification of a Computerized Railway Interlocking," *Formal Asp. Comput.*, pp. 338–360, 1998.

[4] W. Fokkink and P. Hollingshead, "Verification of Interlockings: from Control Tables to Ladder Logic Diagrams," in *FMICS'98*, 1998, pp. 171–185.

[5] S. Vanit-Anunchai, "Modelling Railway Interlocking Tables Using Coloured Petri Nets," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 6116, 2010, pp. 137–151.

[6] K. Winter, W. Johnston, P. Robinson, P. Strooper, and L. van den Berg, "Tool support for checking railway interlocking designs," in *Proceedings of the 10th Australian workshop on Safety critical systems and software*, 2006, pp. 101–107.

[7] A. Mirabadi and M. Yazdi, "Automatic Generation and Verification of Railway Interlocking Control tables using FSM and NuSMV," in *Transport Problems : an International Scientific Journal*, 2009, pp. 103–110.

[8] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano, "A Formal Verification Environment for Railway Signaling System Design," in *Formal Methods in System Design*, 1998, pp. 139–161.

[9] J. F. Groote, S. van Vlijmen, and J. Koorn, "The Safety Guaranteeing System at Station Hoorn-Kersenboogerd," in *Logic Group Preprint Series 121*, Utrecht University, 1995.

[10] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi, "Model Checking Interlocking Control Tables," in *FORMS/FORMAT*, 2010, pp. 98–107.

[11] K. Winter and N. J. Robinson, "Modelling Large Railway Interlockings and Model Checking Small Ones," *Twenty-Fifth (ACSC2003)*, pp. 309–316, 2003.

[12] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *LNCS 2404-CAV '02*, 2002, pp. 359–364.

[13] G. Holzmann, *Spin model checker, the primer and reference manual*, 2003.

[14] A. E. Haxthausen, M. L. Bliguet, and A. A. Kjær, "Modelling and Verification of Relay Interlocking Systems," in *Monterey Workshop*, 2008.

[15] P. James and M. Roggenbach, "Automatically Verifying Railway Interlockings using SAT-based Model Checking," in *AVOCS*, 2010, pp. 141–153.

[16] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *LNCS 1579-TACAS '99*, 1999, pp. 193–207.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in *Computer Aided Verification, 12th International Conference, LNCS 1855-CAV '00*, 2000, pp. 154–169.

Verification of solid state interlocking programs

Phillip James, Andy Lawrence
Faron Moller, Markus Roggenbach,
Monika Seisenberger, Anton Setzer
Swansea Railway Verification Group
Swansea University, Wales, UK

Karim Kanso
Critical Software Technologies
Southampton, England, UK

Simon Chadwick
Invensys Rail Northern Europe
a Siemens company
Chippenham, England, UK

***Abstract*—We report on the inclusion of a formal method into a design process in industry. Concretely, we suggest carrying out a verification step in railway interlocking design between programming the interlocking and testing this program. Safety still relies on testing, but the burden of guaranteeing completeness and correctness of the verification is in this way greatly reduced. We present a complete methodology for carrying out this verification step in the case of ladder logic programs and give results for real world railway interlockings. As this verification step reduces costs for testing, Invensys Rail is working to include such a verification step into their design process of solid state interlockings.**

I. Introduction

Solid state interlockings represent one of many safety measures implemented in railways. In Vincenti's terminology [1], interlockings are *normal* designs: railway engineers have a clear understanding of their workings and customary features, and it is standard practice to design them and to bring them into operation.

The formal method we propose is a verification step between programming the interlocking and the testing of this program. On the one hand we have interlocking programs, their representation in propositional logic, and their semantics in terms of a labelled transition system; whilst on the other hand we have general safety properties expressed in first order logic, their specialization to propositional logic, and their satisfaction relative to the labelled transition system. Both representation and specialization can be automatically derived. The method we suggest is to apply standard model checking approaches and tools to the resulting model checking problem.

We first define interlockings and describe their design exemplified by the GRIP process and the realisation of GRIP's Detailed Design phase at Invensys Rail. We detail our formal method, i.e., the verification step, and compile different technologies upon which the verification can be based, giving comparative results in terms of a case study. We conclude with a brief discussion of related work and future research. This paper summarizes results published in [2]–[10].

II. Designing Solid State Interlockings

In railways systems, solid state interlockings provide a safety layer between the controller and the track. In order to move a train, the controller issues a request to set a route. The interlocking uses rules and track information to determine whether it is safe to permit this request: if so, the interlocking will change the state of the track (move points, set signals, etc.) and inform the controller that the request was granted; otherwise the interlocking will not change the track state. In this sense, an interlocking is like a Programmable Logic Controller (PLC). The standard IEC 61131 [11] identifies programming languages for such controllers, including the visual language ladder logic discussed below.

Interlockings applications are developed according to processes prescribed by Railway Authorities, such as Network Rail's *Governance for Railway Investment Projects* (GRIP) process. The first four GRIP phases define the track plan and routes of the railway to be constructed, while phase five – the detailed design – is contracted to a signalling company such as Invensys which chooses appropriate track equipment, adds control tables to the track plan, and implements the solid state interlocking. It is for part of this phase, namely for the correct implementation of a control table in a solid state interlocking, that our paper offers support in terms of a formal method.

Signalling handbooks (e.g. [12]) describe how to design control tables for the routes of a track plan selected for signalling. Technical data sheets provide information of how to control the selected hardware such as points, signals and track circuits. It is a complex programming task to implement the control tables for the selected hardware elements. For a larger railway station, the resulting program can involve thousands of tightly coupled variables, so thorough testing for safety is a must. To this end, programs are run on a rig which simulates the physical railway, and it can take any number of iterations of testing and debugging for a program to pass all prescribed tests. This testing cycle is cost intensive, as it is hardly automated due to its interactive nature and concerns about the safety integrity of any automated testing environment: the tester has to run the program through various scenarios developing over time. Furthermore, debugging is time consuming as there is little support for producing counter examples.

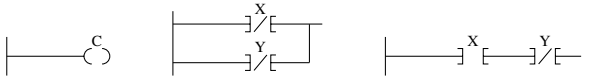
It is at this point that the formal method described below is able to reduce costs in the design process. Rather than testing an interlocking program, we automatically transform the program and the safety property that the test shall establish into a model checking problem. Tool support then allows to automatically check if the property is fulfilled. In case it is not, a counter example is produced, possibly in the form of a trace of controller requests and train movements. This allows the programmer to obtain intelligible feedback. This process is fast and far less involved than testing the program. For these reasons, based on our research, Invensys Rail is working to include such a verification step into their design process of solid state interlockings.

III. From Ladder Logic to Model Checking

A. Ladder Logic

Ladder logic gets its name from its graphical "ladder"-like form (see Fig. 1) reminiscent of relay circuits. Each rung of the ladder computes the current value of an output from the values of one or more inputs in the rung one time step (i.e. one cycle) earlier. A ladder logic program is executed top-to-bottom, and an interlocking executes such a program indefinitely.

A ladder logic rung consists of the following entities. *Coils* represent boolean values that are stored for later use as output variables from the program. A coil is always the right most entity of the rung and its value is computed by executing the rung from left to right. *Contacts* are the boolean inputs of a rung, with *open* and *closed* contacts representing the values of un-negated and negated variables respectively. The value of a coil is calculated when a rung fires, making use of the current set of inputs – input variables, previous output variables, and output variables already computed for this cycle – following the given connections. A horizontal connection between contacts represents logical conjunction and a vertical connection represents logical disjunction. For example:



(a) A coil (b) Disjunction with closed contacts (c) Conjunction with an open and a closed contact

As a running example we model a Pelican crossing, consisting of: two buttons at each side of a road, allowing pedestrians to make a request to cross; and four sets of lights (2 pedestrian lights, pla and plb, and 2 traffic lights, tla and tlb) controlling the flow of pedestrians and traffic. This is modelled by a boolean input variable $pressed$ and 8 variables $plar$, $plag$, $plbr$, $plbg$, $tlar$, $tlag$, $tlbr$, $tlbg$, modelling the aspect of the light, 'r' for 'red', 'g' for 'green'.

We also have two internal variables: req represents whether one of the pedestrian buttons has been pressed in a previous iteration of the program and whether there is already a request to cross; and $crossing$ models the fact that a pedestrian is allowed to cross the road. Fig. 1 presents a ladder logic program for such a Pelican crossing.

B. From Ladder Logic to Propositional Logic

From an abstract perspective, ladder logic diagrams represent propositional formulae. However, the process of obtaining these formulae as described in [2] requires special care to prevent a blow-up in formula size regarding nested disjunctions, which would result in bad performance for CNF translation[1]. This is achieved by traversing the formula from left to right, building up sub-formulae, each of which consisting of a conjunction or disjunction. The efficient use of sub-formulae requires the introduction of auxiliary variables. Fig. 2 shows an example and locations where variables are introduced.

A new variable is introduced for each step in the computation: After every contact x a new variable x_i is introduced (where i is fresh for x), and for each vertical connection (disjunction) a new variable $\vee_j$ is introduced (where j is fresh).

[1]Required when interfacing with theorem provers.

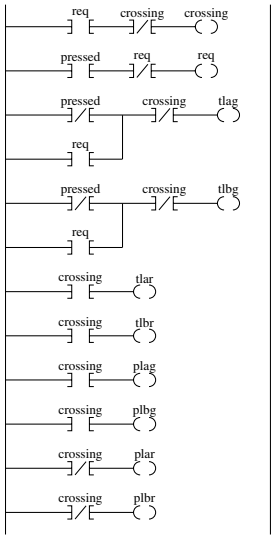


Fig. 1. The ladder logic program for the pelican crossing

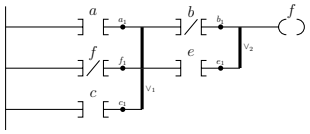


Fig. 2. Try tracing back from coil f: It is clear that the nested disjunction results in the large formula $f' \leftrightarrow (\neg b \wedge (a \vee \neg f' \vee c)) \vee (e \wedge (a \vee \neg f' \vee c))$.

The rung is then broken at each of the intermediate variables, resulting in a simplified ladder. Each rung in the simplified ladder consists of only conjunction or disjunction and at most one negation. By following the above procedure, applied to the ladder in Fig. 2, the below assignments are obtained.

Translating the assignments from (a) below is canonical[2] with respect to the operators, giving the formula in (b):

(a) Assignments of Fig. 2.	(b) Translation of (a).
$a_1 := a$	$(\; a'_1 \leftrightarrow a \quad)$
$f_1 := \neg f$	$\wedge (\; f'_1 \leftrightarrow \neg f \quad)$
$c_1 := c$	$\wedge (\; c'_1 \leftrightarrow c \quad)$
$\vee_1 := a_1 \vee f_1 \vee c_1$	$\wedge (\; \vee'_1 \leftrightarrow a'_1 \vee f'_1 \vee c'_1 \;)$
$b_1 := \vee_1 \wedge \neg b$	$\wedge (\; b'_1 \leftrightarrow \vee'_1 \wedge \neg b \quad)$
$e_1 := \vee_1 \wedge e$	$\wedge (\; e'_1 \leftrightarrow \vee'_1 \wedge e \quad)$
$\vee_2 := b_1 \vee e_1$	$\wedge (\; \vee'_2 \leftrightarrow b'_1 \vee e'_1 \;)$
$f := \vee_2$	$\wedge (\; f' \leftrightarrow \vee'_2 \quad)$

The ladder logic of the Pelican logic in Fig. 1 translates (for readability without the optimization) into the conjunction of

[2]Variables on the rhs of a rung get a prime if they already have been defined in a previous rung.

these formulae:

$$\begin{array}{lcl} crossing' & \leftrightarrow & req \wedge \neg\, crossing, \\ req' & \leftrightarrow & pressed \wedge \neg\, req, \\ tlag' & \leftrightarrow & (\neg\, pressed \vee req') \wedge \neg\, crossing' \\ tlbg' & \leftrightarrow & (\neg\, pressed \vee req') \wedge \neg\, crossing' \end{array}$$

$$\begin{array}{llll} tlar' & \leftrightarrow crossing', & tlbr' & \leftrightarrow crossing', \\ plag' & \leftrightarrow crossing', & plbg' & \leftrightarrow crossing', \\ plar' & \leftrightarrow \neg\, crossing', & plbr' & \leftrightarrow \neg\, crossing' \end{array}$$

C. Ladder Logic Formulae and their Semantics

A ladder logic program is constructed in terms of disjoint finite sets I and C of input and output variables. In our example in Fig. 1, we have $I = \{pressed\}$ and $C = \{crossing, req, tlag, tlbg, tlar, tlbr, plag, plbg, plar, plbr\}$. We define $C' = \{c' \mid c \in C\}$ to be a set of new variables (intended to denote the output variables computed in the current cycle). In addition, we need a function $\mathrm{unprime} : C' \to C, \mathrm{unprime}(c') = c$.

Definition 1 (Ladder Logic Formulae)**.** *A ladder logic formula ψ is a propositional formula of the form*

$$\psi \equiv ((c'_1 \leftrightarrow \psi_1) \wedge (c'_2 \leftrightarrow \psi_2) \wedge \ldots \wedge (c'_n \leftrightarrow \psi_n)$$

such that the following holds for all $i, j \in \{1, \ldots, n\}$:

- $c'_i \in C'$
- $i \neq j \to c'_i \neq c'_j$
- $\mathrm{Vars}(\psi_i) \subseteq I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$

Remark 1. *Note that the output variable c'_i of each rung ψ_i, may depend on $\{c_i, \ldots, c_n\}$ from the previous cycle, but not on c_j with $j < i$, due to the imperative nature of the ladder logic implementation. Those values are overridden.*

Remark 2. *In the formulae extracted from a ladder logic program equivalences $(c'_1 \leftrightarrow \psi_1) \wedge \cdots$ can be replaced by $(c'_1 = \psi_1) \wedge \cdots$. Both formulae are equivalent since for Boolean values b and c the truth values of $b \leftrightarrow c$ and $b = c$ are the same. The use of $\leftrightarrow$ is suitable for the input language of SAT solvers, which require logical formulae (in our example combined with verification conditions) to be checked for satisfiability. The use $=$ is suitable for the input language of model checkers, which require equations defining the variables of the next state in terms of the current one.*

Definition 2 (Semantics of Ladder Logic Formulae)**.** *Let $\{0, 1\}$ represent the set of boolean values and let*

$$\mathrm{Val}_I = \{\mu_I \mid \mu_I : I \to \{0,1\}\} = \{0,1\}^I$$
$$\mathrm{Val}_C = \{\mu_C \mid \mu_C : C \to \{0,1\}\} = \{0,1\}^C$$

be the sets of valuations for input and output variables. The semantics of a ladder logic formula ψ is a function that takes the two current valuations and returns a new valuation for output variables.

$$[\psi] : \mathrm{Val}_I \times \mathrm{Val}_C \to \mathrm{Val}_C$$
$$[\psi](\mu_I, \mu_C) = \mu'_C$$

where

$$\mu'_C(c_i) = [\psi_i](\mu_I, (\mu_C)_{\restriction\{c_i,\ldots,c_n\}}, (\mu'_C \circ \mathrm{unprime})_{\restriction\{c'_1,\ldots,c'_{i-1}\}})$$
$$\mu'_C(c) = \mu_C(c) \textbf{ \textit{if}\ } c \notin \{c_1, \ldots, c_n\}$$

and $[\psi_i](\cdot,\cdot,\cdot)$ denotes the usual value of a propositional formula under a valuation.

D. Labelled Transition Systems

Next we make use of the above to form a labelled transition system representing the ladder logic program.

Definition 3 (Labelled Transition System)**.** *A Labelled Transition System (LTS) M is a four tuple (S, T, R, S_0) where*

- *S is a finite set of states.*
- *T is a finite set of transition labels.*
- *$R \subseteq S \times T \times S$ is a labelled transition relation.*
- *$S_0 \subseteq S$ is the set of initial states.*

We write $s \xrightarrow{t} s'$ for $(s, t, s') \in R$. A state s is called reachable *if $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$, for some states $s_0, \ldots, s_n \in S$, and labels $t_0, \ldots, t_{n-1} \in T$ such that $s_0 \in S_0$ and $s_n = s$.*

Definition 4 (Ladder Logic Labelled Transition System)**.** *We define the labelled transition system $\mathrm{LTS}(\psi)$ for a ladder logic formula ψ to be the four tuple $(\mathrm{Val}_C, \mathrm{Val}_I, \to, \mathrm{Val}_0)$ where*

- *$\mu_C \xrightarrow{\mu_I} \mu'_C$ iff $[\psi](\mu_I, \mu_C) = \mu'_C$*
- *$\mathrm{Val}_0 = \{\mu_C \mid \mu_C$ inital valuation$\}$*

Remark 3. *The standard initial valuation in the railway domain sets all red lights to 1, and all other variables to 0, i.e. this results in exactly one initial state. A variant proceeds as follows: First, all output variables are set to 0 and then all possible transitions are performed. Val_0 is then defined as the set of states obtained after this first transition. In the Pelican crossing example (see Fig. 3 below) this would lead to two initial states rather than one. In both cases, a formula Init characterizes Val_0.*

E. Producing Verification Conditions

In order to guarantee safety, companies such as Invensys ensure through testing that interlockings fulfil certain properties. We formulate them as logical formulae, and call the result *safety conditions*. These conditions are the main example of *verification conditions*, which are formulae, for which we check using our tools whether they hold in an interlocking system. In our setting verification conditions are first-order formulae, with variables ranging over entities such as points, signals, routes, track segments, while referring to predicates. An example of a signalling principle is the formula

$$\begin{array}{l} \forall rt, rt' \in \mathrm{Route}. \forall ts \in \mathrm{Segment}. rt \neq rt' \\ \quad \to (\mathrm{part\_of}(ts, rt) \wedge \mathrm{part\_of}(ts, rt')) \\ \quad\quad \to \neg(\mathrm{routeset}(rt) \wedge \mathrm{routeset}(rt')) \end{array}$$

expressing the property: *for all pairs of routes that share a track segment, at most one of them can be set to proceed*.

Note there are two kinds of predicates: *State* and *Topology*. State predicates express the state of entities at a given time. E.g. $\mathrm{routeset}(rt26)$ expresses that route $rt26$ has been set. These predicates will unfold into variables in the ladder logic program, so in the previous example the predicate would—depending on the actual naming scheme—unfold to the variable $rt26ru$. *Topology* predicates express meta information relating to the topology of the railway yard.

E.g. $\text{part\_of}(ts54, rt26)$ expresses that the track segment $ts54$ is part of route $rt26$. These predicates unfold to *true* or *false*, depending on whether the property holds; thus, the previous example unfolds to *true* when $ts54$ is actually part of $rt26$, otherwise *false*.

Some topology predicates are atomic and stated explicitly as true or false for given arguments. Other predicates can be computed in terms of these atomic predicates. E.g., signal $ms1$ is a main signal guarding access to route rt, if there exists track segments $ts1$ and $ts2$ such that $ts1$ is before route rt, $ts1$ is connected with $ts2$, $ts2$ is part of the route rt, and $ms1$ is located directly between $ts1$ and $ts2$. This can be expressed as follows:

$$\begin{aligned}&\text{route\_main\_signal}(ms1, rt) \leftrightarrow \exists ts1, ts2 \in \text{Segment}.\\ &\quad \text{before}(ts1, rt) \wedge \text{connected}(ts1, ts2) \wedge \text{part\_of}(ts2, rt)\\ &\quad \wedge \text{infrontof}(ts1, ms1) \wedge \text{inrearof}(ts2, ms1)\end{aligned}$$

In [2], [6] Kanso introduced a translation of such formulae to propositional formulae which then can be verified using SAT solving or model checking. He took the following steps:

(1) Expressed the topology as a Prolog program, which determined the truth value of the topology predicates. It consisted of clauses such as $\text{mainsignal}(ms1)$ ($ms1$ is a main signal), $\text{infrontof}(ts0a, ms1)$ (signal $ms1$ is in front of track segment $ts0a$). The above predicate $\text{route\_main\_signal}(ms1, rt)$ is defined in Prolog as:

$$\begin{aligned}&\text{route\_main\_signal}(ms1, rt) :-\\ &\quad \text{before}(ts, rt), \text{connected}(ts, tss),\\ &\quad \text{part\_of}(tss, rt), \text{infrontof}(ts, ms1),\\ &\quad \text{inrearof}(tss, ms1).\end{aligned}$$

(2) Translated using standard techniques from logic the formula into prenex form, i.e. a formula starting with a block of quantifiers followed by a quantifier free formula.

(3) Now $\forall x \in A.\varphi(x)$ is replaced by $\varphi(a_1) \wedge \cdots \wedge \varphi(a_n)$ and $\exists x \in A.\varphi(x)$ by $\varphi(a_1) \vee \cdots \vee \varphi(a_n)$, where $a_1, \ldots, a_n$ are the elements of set A in the topology. φ is now instantiated to closed instances. Therefore the topological predicates evaluate to truth values true or false, which can then easily be omitted from the formula. Safety formulae can usually be translated into universally quantified formulae in prenex normal form[3]. The universally quantified formula is replaced by conjunctions, where most conjuncts reduce to false, since topology predicates such as $\text{connected}(ts1, ts2)$ are false for most choices of arguments. Finally state predicates are replaced by the Boolean variables of the ladder logic. In case of safety conditions we obtain a conjunction of instantiations of ψ. Since safety conditions usually become conjunctions, the validity of the conjuncts can be checked separately for validity. This allows to identify problems relating specific objects of the railway yard.

A typical verification condition for our Pelican crossing example would for instance ensure that the traffic lights and the pedestrian lights are not green at the same time:

$$\varphi \equiv (tlag \wedge tlbg \wedge \neg plag \wedge \neg plbg) \vee (\neg tlag \wedge \neg tlbg \wedge plag \wedge plbg)$$

[3] $\forall x_1 \in A_1, \ldots, x_n \in A_n.\varphi(x_1, \ldots, x_n)$, where φ is quantifier free.

F. The Model Checking Problem

Definition 5 (Safety Conditions for a Ladder Logic Program)**.** *Given a ladder logic formula ψ over the variables in $I \cup C$ a **verification condition** is a propositional formula formed from the variables in $I \cup C \cup C'$.*

Definition 6 (The Verification Problem for Ladder Logic Programs)**.** *We define the verification problem for a ladder logic formula ψ for a verification condition ϕ*

$$\text{LTS}(\psi) \models \phi$$

iff for all triples μ_C, μ_I, μ'_C such that $\mu_C \xrightarrow{\mu_I} \mu'_C$ and μ_C is reachable in $\text{LTS}(\psi)$, we have $[\phi](\mu_C, \mu_I, \mu'_C) = 1$.

Note that in most cases, as in our Pelican crossing example, the verification condition ϕ only consists of variables in C, therefore the model checking problem simplifies to considering individual states, i.e. whether $[\phi](\mu_C) = 1$ at all times. Fig. 3[4] shows the labelled transition system for the Pelican crossing example. We have included one unreachable state in which both `required` and `crossing` are true.

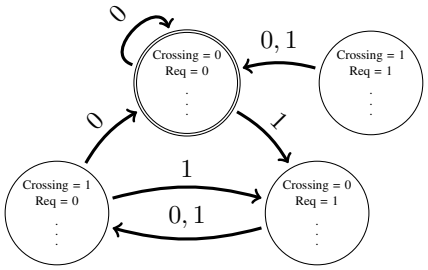


Fig. 3. Pelican crossing transition system

G. Model Checking Approaches

Target technology for the first three algorithms is SAT-solving; in the algorithms, execution terminates after a "return" statement has been performed.

1) Bounded Model Checking (BMC): BMC, see, e.g., [13], restricts the depth of the search space. Let the formulae ψ_n^{Init}, $n \geq 1$, be unrolled transition relations which encode n steps with ψ from an initial state of the LTS. The following algorithm explores the LTS to a depth of up to K steps (we assume that ϕ uses the variables concerning the last state):

```
if ¬(Init → φ) satisfiable, return error state
n ← 1
while n ≤ K do
    if ¬(ψ_n^Init → φ) satisfiable, return error trace
    n ← n + 1
return "K-Safe"
```

As BMC produces a counter example trace if the verification fails, it is especially interesting for debugging purposes.

[4] The transition labelled 0,1 is in fact two transitions, one labelled with 1 and the other labelled with 0.

2) Inductive Verification (IV): IV checks if an over-approximation of the reachable state space is safe. In the following algorithm we assume that ϕ uses the variables concerning the current state and ϕ' those concerning the last state:

```
if ¬(Init → φ') satisfiable, return error state
if ¬(ψ ∧ φ → φ') satisfiable, return pair of error states
return "Safe"
```

The over approximation happens in the second line of the algorithm: here one considers all safe states rather than the *reachable* ones. This idea makes IV a very efficient approach involving at most two calls to a SAT solver [2], [6].

3) Temporal Induction (TI): TI, see, e.g, [14], combines BMC and IV to allow for both: complete verification and counter example production. Let ψ_n be the unrolled transition relation encoding n steps with ψ, let LF_n be a formula encoding that all n states of a sequence of states are pairwise different and $safe_n$ be a formula encoding that all these states fulfil the verification condition, $n \geq 0$. Define $Base_n \equiv$ Init $\wedge\ \psi_n \rightarrow \phi$ and $Step_n \equiv \psi_{n+1} \wedge LF_{n+1} \wedge safe_n \rightarrow \phi$, $n \geq 0$, where ϕ uses the variables concerning the last state.

```
n ← 0
while true do
    if ¬Base_n satisfiable, return error trace
    if ¬Step_n unsatisfiable, return "Safe"
    n ← n + 1
```

4) Stålmarck's Algorithm: This algorithm has been developed and patented by Stålmarck [15]. It usually works well on industrial problems as they are often of considerable size, but with a simple underlying structure. This is due to its ability to merge the conclusion of branches in a proof tree which can be seen as a form of learning. Its underlying theory was influenced by sequent calculus and semantic tableaux which inspired the branch and merge dilemma rule and the simple proof rules respectively. The algorithm makes use of equivalence classes in the form of data structures known as triplets.

5) Optimization via Slicing: Usually, the verification condition ϕ does not use all variables of the ladder logic formula ψ. This opens up the possibility to slice ψ with respect to ϕ, i.e., to compute a formula ψ_ϕ with $\psi \models \phi \Leftrightarrow \psi_\phi \models \phi$ where ψ_ϕ involves fewer variables and rungs than ψ. [16], [17] present an algorithm to compute ψ_ϕ, [4], [9] give a correctness proof. Here is the sliced ladder logic program of the Pelican crossing example for the condition $(tlag \vee tlar) \wedge \neg(tlag \wedge tlar) \wedge (tlbg \vee tlbr) \wedge \neg(tlbg \wedge tlbr)$:

$$\begin{aligned} crossing' &\leftrightarrow req \wedge \neg crossing, \\ req' &\leftrightarrow pressed \wedge \neg req, \\ tlag' &\leftrightarrow (\neg\, pressed \vee req') \wedge \neg\, crossing' \\ tlbg' &\leftrightarrow (\neg\, pressed \vee req') \wedge \neg\, crossing' \\ tlar' &\leftrightarrow crossing', \\ tlbr' &\leftrightarrow crossing' \end{aligned}$$

Such slicing can be applied as a pre-processing step for all four approaches discussed above.

H. Excluding False Positives by Invariants

When verifying concrete examples, often false positives were obtained. When discussing these counter examples with railway experts, one obtains usually that these examples do not occur because a certain combination of values for variables is not possible. This means that a certain invariant was violated. We identified [2] two kinds of invariants, *physical* invariants and *mathematical* invariants. *Physical invariants* are due to the fact that certain combinations of input variables are physically impossible. An example is a three way switch, which is modelled by 3 variables where each variable i indicates whether the switch is in position i or not.[5] It is physically impossible for this switch to be in two positions simultaneously. Physical invariants need to be carefully investigated by domain experts. One example could be a paper clip falling into a three way switch, which connects then two contacts, and one might want the railway yard to be safe even if a paper clip has falled into the switch.

Mathematical invariants. When using IV one might obtain states which violate the safety condition, but are not reachable from the initial state. In this case one can identify invariants, which hold in all reachable states but not in the false positive. In many cases one can prove now using the tool that the invariant holds in all cases, and then prove again using the tool that the verification condition holds provided the invariant holds.

I. Graphical Representation

In order to investigate counter examples a graphical representation of the error states was given. For our prototype Kanso [2], [6] developed a latex document, which contained a scheme plan with signals sets of points and routes, together with tables listing the state of all variables in question. The state of signals (red or green) and points and of all tables listed was determined by macros. It was now easy to compute from an error state a document setting these macros to the values in this state, and therefore present an easy to view document.

IV. Technology & Case Studies

A. Sat-Solving with open software

An initial—successful—feasibility study was conducted using the open-source OKLibrary as underlying SAT solving framework to automate IV in order to establish safety properties. To this end, we used the Dimacs format as a target language. Note that this requires a representation in CNF.

Extending this implementation, we produced a framework of automatic translations of the formulae ψ, written in Haskell (about 8000 lines of code), and ϕ, written in Java (about 1000 lines of code), into the formulae required for the algorithms BMC, IV, and TI. As target format we chose TPTP [18], which is the input language of the Paradox tool [19]. Internally, the open source tool Paradox is based on the SAT solver Minisat [20], which is open source as well. Using Paradox has the advantage that the tool takes care of the translation into Dimacs format. The framework also includes a Haskell implementation of slicing (about 500 lines of code).

[5] One could easily model it by 2 variables; however having 3 variables makes it easier to compute the next state from the current state.

Using this framework, experiments on our Pelican crossing example with the above verification condition showed: with BMC the program is K safe for all $K \geq 0$ we tried; with IV, we obtain a pair of error states; TI gives the result "Safe". This example demonstrates that though IV is sound, it is not complete.

B. The SCADE Suite as an Industrial Tool

For comparison, we applied a tool widely used in Industry, where however no control over the method applied is available. In *SCADE* (Safety Critical Applications Development Environment) [21] programs are verified using the *SCADE* language and Prover Technology based on Stalmarck's algorithm. The program to translate ladder logic programs into *SCADE* language is based on the framework described above, it has a length of approximately 8000 lines of Haskell code [5].

The *SCADE* language is based on the synchronous dataflow language Lustre [22]. The flows which constitute a Lustre program are infinite sequences of values which describe how a variable changes over time. Flows are combined together to form nodes which can be seen as the Lustre equivalent of a function or procedure. There are two main temporal operations which can be applied to flows:

- The operator `pre` allows one to speak about the previous value of a flow.
- The operator `->` allows one to speak about the initial value of a flow and its successive values.

The following is the result of the automatic translation of the pelican crossing ladder logic to *SCADE*.

```
node PelicanLadderLogic1(pressed: bool)

returns (req, crossing, tlag, tlar, tlbg, tlbr, plag,
                        plar, plbg, plbr: bool)

let crossing = false -> pre req and (not (pre crossing));
    req = false -> (not pre req) and pressed;
    tlag = false -> ((not pressed) or req) and (not crossing);
    tlbg = false -> ((not pressed) or req) and (not crossing);
    tlar = true -> crossing;
    tlbr = true -> crossing;
    plag = false -> crossing;
    plbg = false -> crossing;
    plar =  true -> not crossing;
    plbr =  true -> not crossing;
tel
```

C. Industrial Case Study

Using the approaches described above we automatically translated real world railway interlockings and safety properties into the Dimacs format (for IV), the TPTP language (for BMC, IV, and TI) and the *SCADE* language. The verification results gained have been positive. For every safety condition the tools have either given a successful verification, or a counter example (trace). All results have been obtained within the region of seconds.

In the following we report on the verification of a small, real world interlocking which actually is in use on the London Underground. The ladder logic program consists of approximately six hundred variables and three hundred and fifty rungs. Concerning typical verification conditions, slicing reduces the number of rungs down to 60 rungs, i.e., the program size is reduced by a factor of 5. All experiments reported have been carried out on a computer with the operating system Ubuntu 9.04, 64-bit edition, an Intel Q9650, Quad core CPU with 3GHz, and a System Memory of 8GB DDR2 RAM.

1) Evaluation with an Open Source Tool: The first condition encodes that if a point has been moved, it must have been free before. Here, the verification actually fails. IV yields a pair of states within 0.75s, while BMC produces an error trace of length 3 in 0.81s, TI produces the same trace. The rail engineers were able to exclude this counter example as a false positive. By adding justifiable invariants we could exclude this false positive. The second condition excludes that the program gives an inconsistent command, namely, that a point shall be set to normal and to reverse at the same time. IV proves this property in 0.71s; BMC yields K-safety for up to 1000 steps, after which we ran out of memory; BMC on the sliced program is possible up to 2000 steps; TI does not terminate, neither for the original nor for the sliced version. Our experience is that IV can deal with real world examples. Slicing yields an impressive reduction of the size of the ladder logic program. It is beneficial when producing counter examples with BMC as it reduces the runtime and also helps with error localization.

2) Verifying the Industrial Case Study using SCADE: All above safety conditions take times less than 1s [5]. We attempted the verification of 109 safety conditions out of these 54 were valid and 55 produced counter examples. The latter are false positives and were eliminated by adding invariants as described above. The total time for the verification and production of counter examples for all of these safety conditions was under 10 seconds. This may be in part due to some support for multi-core processors allowing the *SCADE* suite to dispatch multiple verification tasks efficiently. Generally, in the process of removing false positives approximately one hundred invariants were added. Overall, this shows that *SCADE* is a viable option for the verification of railway interlockings.

V. Conclusion

The overall conclusion is that the verification step described works out: the required translations can be automated, the current tools scale up to real world problems, the gained benefits are convincing enough for the company Invensys to change its practice. In terms of the underlying proof technology, it is a matter of taste / philosophy / further constraints if one wants to employ open software tools or a commercial product.

Our work on verifying ladder logic programs has been inspired by [16], [17]. Alternative approaches include [23] who apply timed automata and UPPAAL or [24] who present a development framework for ladder logic, including verification by port-level simulation. Our contribution is to put known verification approaches into the context of a concrete engineering problem and, by providing a prototypical implementation, demonstrating that they work.

Putting the context even wider, in his PhD thesis [3] Kanso shows how to fully verify railway interlockings by interactive theorem proving. This work greatly reduces the gap between formal verification of safety and safety in the real world.

REFERENCES

[1] W. G. Vincenti, *What engineers know and how they know it*. The Johns Hopkins University Press, 1990.

[2] K. Kanso, "Formal verification of ladder logic," 2010, MRes Thesis, Swansea University.

[3] ——, "Agda as a platform for the development of verified railway interlocking systems," 2012, PhD Thesis, Swansea University.

[4] P. James, "SAT-based model checking and its applications to train control software," 2010, MRes Thesis, Swansea University.

[5] A. Lawrence, "Verification of railway interlockings in SCADE," 2011, MRes Thesis, Swansea University.

[6] K. Kanso, F. Moller, and A. Setzer, "Automated verification of signalling principles in railway interlocking systems," *ENTCS*, vol. 250, pp. 19–31, 2009.

[7] K. Kanso and A. Setzer, "Specifying railway interlocking systems," in *PreProceedings of AVoCS'09*, 2009, pp. 233 – 236.

[8] ——, "Integrating automated and interactive theorem proving in type theory," in *Proceedings of AVOCS 2010*, 2010.

[9] P. James and M. Roggenbach, "Automatically Verifying Railway Interlockings using SAT-based Model Checking," in *Proceedings of AVoCS'10*. Electronic Communications 35 of EASST, 2010.

[10] A. Lawrence and M. Seisenberger, "Verification of railway interlockings in SCADE," in *Proceedings of AVOCS 2010*, 2010.

[11] IEC, "IEC 61131-3 edition 2.0 2003-01. international standard. programmable controllers. part 3: Programming languages," January 2003.

[12] M. Leach, Ed., *Railway Control Systems: a sequel to Railway Signalling*. A & C Black, 1991.

[13] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," in *Formal Methods in System Design*. Kluwer Academic Publishers, 2001, p. 2001.

[14] N. Een and N. Sörensson, "Temporal induction by incremental SAT solving," *ENTCS*, vol. 89, no. 4, 2003.

[15] G. Stålmarck, "System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula," 1994, US Patent: 5,276,897.

[16] J. Groote, J. Koorn, and S. Van Vlijmen, "The safety guaranteeing system at station Hoorn-Kersenboogerd," in *Compass'95*. IEEE, 1995.

[17] W. Fokkink and P. Hollingshead, "Verification of interlockings: from control tables to ladder logic diagrams," in *FMICS'98*, 1998.

[18] "The TPTP problem library for automated theorem proving," `http://www.cs.miami.edu/ tptp/`.

[19] K. Claessen, "New techniques that improve mace-style finite model finding," in *CADE-19*, 2003.

[20] "Minisat," `http://minisat.se`.

[21] P. Abdulla, J. Deneux, G. Stålmarck, H. Argen, and O. Akerlund, "Designing safe, reliable systems using SCADE," in *Springer LNCS 4313*, 2006, pp. 115–129.

[22] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: a declarative language for real-time programming," in *POPL'87*, 1987.

[23] B. Zoubek, J.-M. Roussel, and M. Kwiatowska, "Towards automatic verification of ladder logic programs," in *CESA'03*. Springer, 2003.

[24] K. Han and J. Park, "Object-oriented ladder logic development framework based on the unified modeling language," in *Computer and Information Science*. Springer, 2009.

Modelling Functionality of Train Control Systems using Petri Nets

Michael Meyer zu Hörste
and Hardi Hungar
German Aerospace Centre (DLR)
Institute of Transportation Systems
Lilienthaplatz 7, 38108 Braunschweig, Germany
Email: {Michael.MeyerzuHoerste,Hardi.Hungar}@dlr.de

Eckehard Schnieder
Technical University Braunschweig
Institute for Traffic Safety and Automation Engineering
Langer Kamp 8, 38106 Braunschweig, Germany
Email: E.Schnieder@tu-bs.de

***Abstract*—Railway safety systems are highly complex systems with respect to functionality as well as dependability. The new European Train Control System (ETCS) as one part of the European Rail Traffic Management System (ERTMS) is the example presented here. A formal model using Coloured Petri Nets (CPN) was prepared by using the existing ERTMS/ETCS specification as a basis. The applied method is an integrated event- and data-oriented approach, which shows the different aspects of the system on their own Petri Net levels. The model comprises three sub-models with a model of the environment developed next to the onboard and trackside systems. This environment model covers all the additional systems connected through the system interfaces, examples of which are interlocking or regulation. Starting from a net representing the system context, the processes of the onboard and trackside sub-systems were modelled. Here, the different operations and processes are visualized in the form of scenarios, which in turn have access to additional refinements representing specific functions.**

I. Introduction

Complex systems as train control systems are specified by many functional, safety-related or other requirements. Showing completeness and consistency of these requirements is a quite difficult task. In the case of ETCS many documents - so-called Subsets - have been written for different purposes. Characteristics of these documents are that they specify different aspects, are based on each other, and have different objectives [1], [2]. From this follows that there are separate documents for specific sub-functions and sub-systems which specify a certain segment of the requirements made on the system, a phenomenon which is most apparent with the central systems which immediately adjoin the interoperable interfaces. The documents can be seen as forming a specification network interlinked by references and cross-references. All this means that, in order to finalize specification work and set about implementation, the specification has to be proved to be fully consistent, and system operability has to be verified; it should also be checked that the system suits the operational conditions of the different railway operators and countries. In the past, a multitude of highly diverse methods and means of description, employing specific computer tools or even manual operations, were used to meet these requirements.

The basic architecture of ETCS exhibits an air gap between the trackside and the onboard subsystems. In one typical equipment configuration, the trackside is realised by a Radio Block Centre (RBC) which sends messages by radio to the Onboard Unit (OBU). The model developd follows this system structure and adds a third part, which is the common system environment. More details can be found in [3]. The Fig. 1 shows the structure.

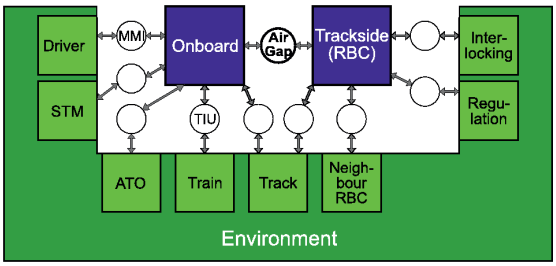


Fig. 1. Example train control system architecture

Central aspects were the air gap and the EuroBalise interface, which were to be modelled with as much detail as possible and in compliance with the System Requirements Specification [1].

II. Some Definitions

A. The term 'model'

A model can in a general way be characterised by three properties: a model represents part of the reality, and of that reality those features that are of relevance for the immediate purpose of the model; other features that are without relevance for the purpose of the model are only presented in a reduced form. The model serves a specific purpose; this characteristic of a model is referred to as pragmatics. A model can serve to describe certain conditions, it can provide insights, or it can replace a real system.

B. Means of description, method and tool

The GMA sub-committee 1.8.1 "Standardisierte Beschreibungsmittel in der Automatisierungstechnik" (Standardised means of description in automation engineering) provides the following definitions for the terms 'means of description', 'method' and 'tool' [4]:

Means of description: A means of description describes graphically certain conditions for visual perception and storing. Means of description are alphanumeric signs, symbols and other graphic elements of representation (semiotics,) and also conventions on how these can be combined (syntax). Assigned to the different elements of representation, their possible combinations and allocations are specific conditions and concepts from a certain context, which may be specified in a more or less detailed and formal way (semantics).

The following distinctions are made:

Formal means of description: Has a mathematical basis and a defined and complete syntax.

Semi-formal means of description: Has a defined and complete syntax, not, however, a mathematical basis.

Informal means of description: Also possesses the characteristics of a means of description (semiotics, syntax, semantics), these are, however, not always complete.

The means of description used in modelling ERTMS/ETCS are coloured and hierarchic Petri nets, since the aim is to examine whether these offer the possibility of using one uniform means of description for the entire development cycle, starting with the specification through to implementation. Above and beyond that, Petri nets provide the required capacity that allows different methods to be used during one single phase of the development cycle and also phase-specific methods [5].

Method: A method is a procedure, systematic both in terms of the point in question and the purpose, following a set of principles and designed to produce insights and practical results.

In the project presented here, an integrated method was used, which has both a data and event-related orientation. It provides for visualization of the sub-system processes, but also of the operational processes in the form of scenarios, and of the functions in individual nets.

Tool: Designed to assist man in or during the production of results. Today, the term 'tool' is normally understood to mean 'realised by computer systems (hardware / software)'.

Following comparative investigations and studies, Design/CPN was selected as tool for this project. The decision was not least taken because of the fact that it provides for a reachability analysis [5], [6].

III. Model Structure

A. Principles of the Net Structure

For ERTMS/ETCS modelling it was decided to visualize system context, sub-system process, operational processes in the form of scenarios and functions in an integrated manner. For this purpose, nets are decomposed to reflect these four aspects in four levels (see Table I).

TABLE I. The different levels of nets within the model

Level	Content
Context	System and relation to system envirnoment (Architecture)
Process	System and interfaces (Interfaces)
Scenario	Reactive sequence of messages and events (Event-Sequence)
Function	Details of functional steps with respect to an event (Functionality)

B. Overall Model / Context Net

The system context is depicted on the uppermost level. This net comprises the two modelled sub-systems onboard system and RBC. All the other sub-systems are comprised in the environment, they may, however, be further refined during a later phase of modelling work.

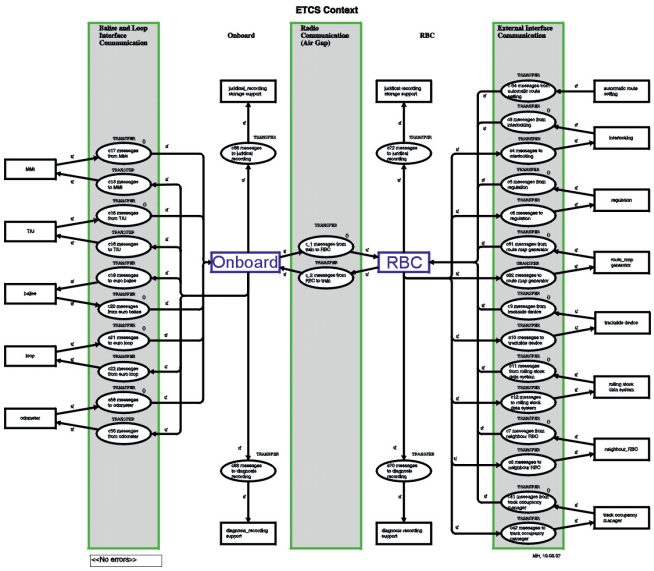


Fig. 3. Petri Net model level 1: System context

This net corresponds to the formal representation of the system architecture in Fig. 1. On this level, all the interfaces are defined as uni-directional channels.

C. Nets on the Process Level

The next level is formed by the nets of process visualization. This level defines what scenarios can be passed in what sequence.

In a central position are the transitions and places that provide additional application logics. To the right and to the left are two dark-grey boxes, which accommodate the interfaces to the outside. Messages sent to a receiver or received by a sender are combined in one place. At the train end the right-hand box, and at the RBC end the left-hand box, is reserved for communications between train and control system.

D. Scenario Nets

The different scenarios are refined with two more aspects. The interfaces are disintegrated until individual messages are singled out. For each message there is a transition, referred to as driver, which is responsible for translating the message from the general data type "message" into the required

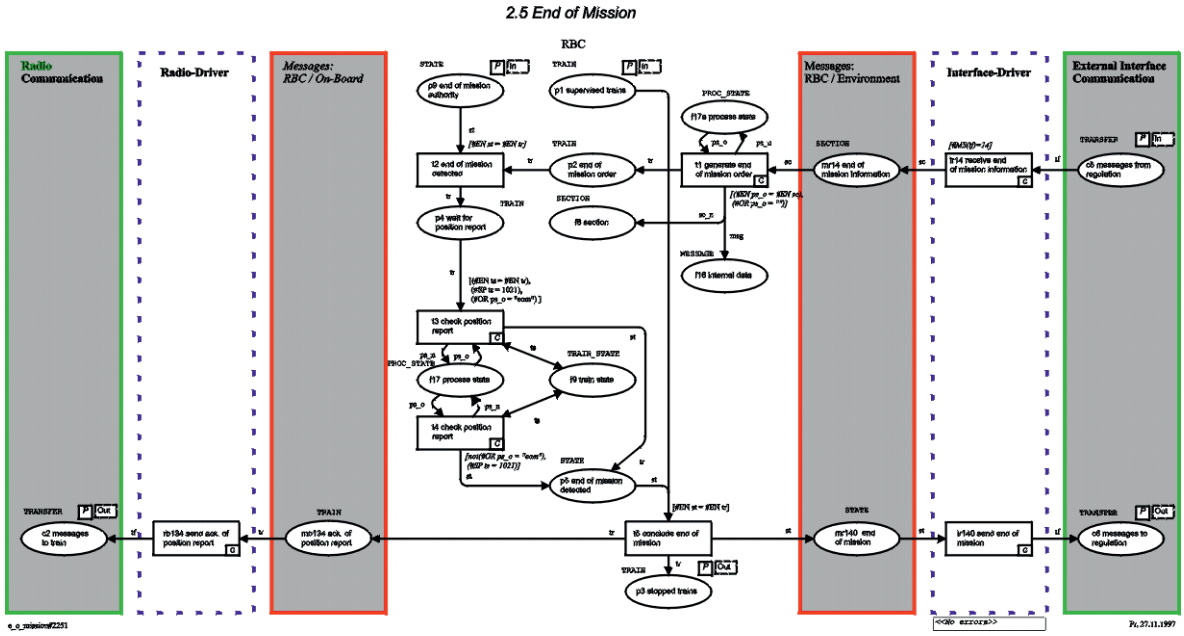


Fig. 2. Petri Net model level 3: Scenario

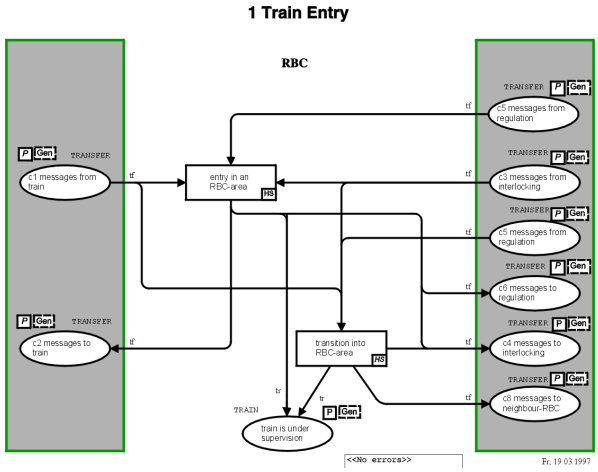


Fig. 4. Petri Net model level 2: Process

context specific type. In this context, messages are defined strictly in compliance with the definitions in chapters 7 and 8 of the System Requirements Specification [1]. In the nets themselves, the sequence of events is shown. The possibility of parallel event sequences as a function of specific initial situations and stimulations is given due consideration. Used as a basis here is SRS [1] and the related low-level documents.

These nets comprise three areas: the application logics is again located in the centre, while the two boxes with the interfaces are located at the outer left and right. Next to the interfaces is a box designated radio driver and interface driver, respectively. All transitions in these boxes start with 'send' or 'receive'; they comprise the receive or send logics for a specific message. This is why between these boxes and the application logics there are one or several more dark-grey boxes, providing one place for each individual message. The name of these boxes starts with "messages". This overall structure is common to all scenario nets.

E. Functional Nets

The functions, if explicitly modelled, form a net level of their own, and are visualized in the following way.

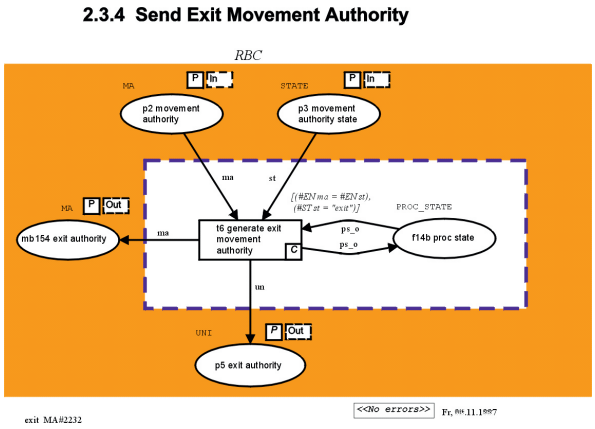


Fig. 5. Petri Net model level 4: Function

The nets on this level are much simpler in structure: they only show the transition refinements. This is why they do not have any interfaces with external systems, even if the transition itself has such interfaces. Communication is safeguarded by the higher (scenario) level net. The net logics is accommodated in a box delimited by a dashed line; the inputs and outputs of the transitions of the higher-level net are outside this box.

IV. Results

A. Model Complexity and Performance

ERTMS/ETCS system modelling has to date proceeded to the degree of complexity given in Table II. As modelling aims at providing a detailed visualization of the air gap, the environment model only serves to produce stimulations, which is why the environment model was not taken beyond the required abstraction level.

TABLE II. Model complexity

	Onboard	RBC	Environment
Number of nets	75	87	6
Net elements	1,075	1,411	97
Places	734	954	65
Transitions	341	457	32
Hierarchy levels	7	7	3
Lines of code (incl. comments)	15,500	12,500	0

The model performance can be subdivided into two groups. On the one hand, modelling itself has achieved a number of aims: the SRS has been modelled in a formal way, the interface that has a central role to play for interoperability, i.e. the air gap between track and train, has been visualized, and the operational processes have been shown in the form of scenarios. A second group is represented by the simulation, which implies both simulation of the operational processes and simulation of the supervision functionality.

One may compare the modeling approach chosen here, namely to use Petri nets, to one emplying state machines from the UML. Both share the idea of introducing hierarchy to offer views at different levels of detail. A main difference lies in the form in which the communication mechanism is represented. The Petri net model represents communications (or communication relations) explicitly in a graphical form via places and tokens. This is in particular helpful when communication is a major concern, as it is the case with the air gap. State machines would rather use events, i.e., non-graphical elements. This gives per se more readable results when entities partake in many activities. Here, this occurs at level 4. Our Petri net model employs "shared places" to remain readable. These are places like the one on the right in Fig. 5, which appear in more than one net.

B. Quality Assurance of the Specification

Validation of the model is, of course, an important aspect. The precise goal of the validation will depend on the purpose for which the model is to be used. Manual techniques like inspections or reviews [8], [9] common for program verification can be transferred to semi-formal and formal models.

These manual techniques can be complemented by model animations and simulations, or by specific analyses. Some reachability analyses have already been done, in which individual scenarios and sequences of scenarios were used as examples. See Fig. 6 for an example visualisation of the results. This figure shows an occurancy graph of a part of the model for 13 different external messages A to M coming in. Generally can be seen, that normally the message comes at a defined state and after a number of events - partially in a fixed sequence, partially parallel - the systems reaches one defined state. The important difference is the message K, where two branches appear, which never converge again. The OG shows here a failure either in the model or in the specification. This exmaple shows, how analysis techniques can be used for quality assurance of the model and the specification, too

V. Use of the Petri Net model

A. Safety Standards / Certification Support

The formal model can assist in furnishing the proof of safety standards. Formal description of the system at the same time provides a clear and unequivocal definition of the system behaviour. On this basis system implementation can be tested with reference to the model, and the system behaviour can be checked for given conditions and also in an abstract manner, in the form of an analytical procedure.

B. Use for Test and Validation of Products

In this respect, the model will serve as a reference for the behavior of the product. There are different forms of relation between model and product behavior.

1) Abstract observer: Each of the different levels of the Petri net can be viewed as an abstraction of the real system. A formal definition of the abstraction relation will map concrete traces to abstract ones (more generally, it may relate the trace sets). Then, any animation, simulation, test, or field observation of an implementation yields traces which can be checked for consistency with the model. This is a standard use case for any kind of model which has an operational semantics.

On the other hand, such observations on an implementation may also be seen as a check for the correctness of the model. In system development, abstract specifications done in early design steps are often not correct in a strict interpretation of this term. For instance, they may be too restrictive (overspecification) or lack important detail (error handling, effects of low-level timing). Such discrepancies, if uncovered, can be used to correct or complete the model. Checking the observer relation is thus a means to arrive at a consistent development documentation.

2) Test construction: The scenario and functional nets provide (as indicated above) abstract view of the behavior of the implementation. These can obviously also be used as skeletons of test cases. To turn them into applicable test cases, the skeletons will have to be parameterized, translated to technical interfaces of the unit under test, and turned into executable scripts with stimuli specifications and pass/fail criteria. In other words, one may derive a test specification from the Petri net model by systematically covering the model.

3) Supporting hardware-in-the-loop tests: The approach sketched above is a way in which a single activity, namely test derivation, in standard development processes may be improved. There are more ambitious scenarios, in which test execution and the development itself might be modified. Thes require further extensions to the model.

Presently, simulations of the model are made off-line, and timing is discrete in the form of discrete steps of sequences of events. Real-time simulations of models presuppose that in modelling the system, aspects of the real-time mode are

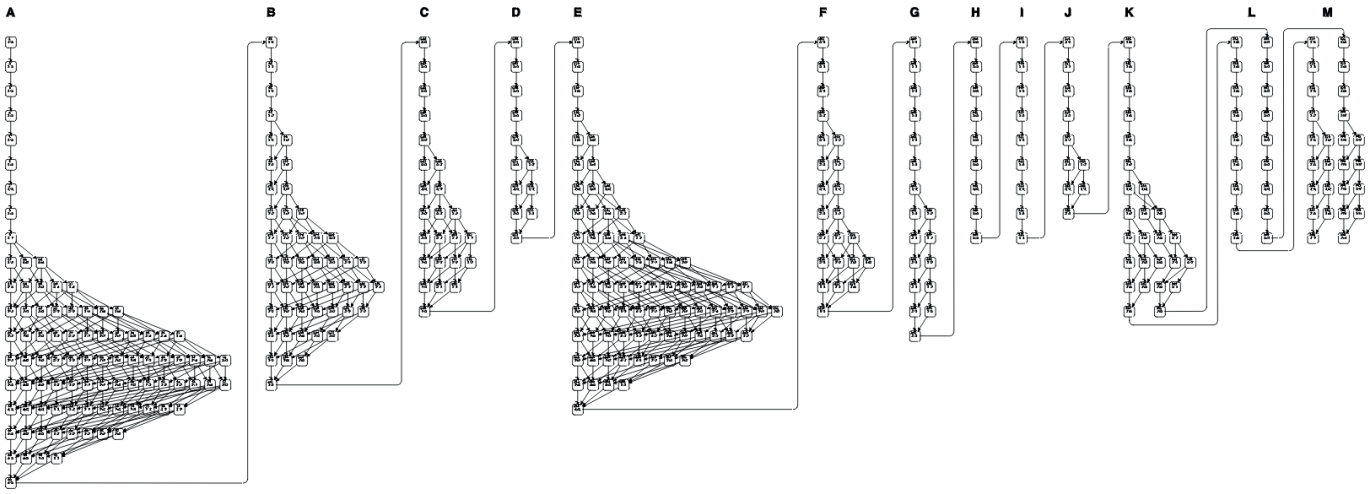


Fig. 6. Occurence Graph of Transition Level 1 to Level 2/3

given due consideration. I.e., the model needs a real-time interpretation, for which some additions to the model will be necessary. Also, a real-time compatible tool for simulating the model is needed.

Adequate modelling and hardware provided, the simulatable model can support tests in the form of hardware-in-the-loop tests. For such tests, part of the real system is replaced by a simulation using suitable hardware, and one or several parts of the system are linked in the form of their implementation. This procedure allows hardware components to be tested individually.

An overview about different simulations for validation and hardware-in-the-loop tests are given in the table III taken from [3]. "sim." means here simulated. The bold numbers show the target system of the test.

TABLE III. SIMULATIONS AND TESTS

onboard		**trackside**		**Aim**
sim.	**real**	**sim.**	**real**	
1-2	0	**1**	0	Validation of oeprational procedures
0	**1**	1	0	Test and validation of onboard subsystem
1-n	0	0-m	**1**	Test and validation of trackside subsystem
0-n	**1**	0-2	**1**	Validation of Interoperability
0	**1**	0	**2**	trackside handover
2-n	0	0-m	**1**	Stresstest

4) Interface Generators: A simulatable model can be used to supply the interfaces of a real or simulated system with stimulations. Possible approaches are presented in [7].

5) Code Generation: Generally executable models can be transformed into software code. Analysed are the following three options of generating executable codes:

1) Automatic source code generation, which is followed as a second step by conventional compilation.
2) Compilation of the Petri net itself to produce an executable code.
3) Execution of the net itself in the form of a system. Within the meaning of the definition of the term 'model' this implies that the system is substituted by a model and its hardware.

VI. CONCLUSION

On the basis of the System Requirements Specification and the relevant documents [1] an ERTMS/ETCS model was developed. This model visualizes the system in the form of three Petri net models. The onboard and trackside systems together form the core which is embedded in a model of the environment. The two core models reflect the aspects of the sub-system context, of the processes proceeding within the sub-systems, of operational processes in the form of scenarios and specific functions. In developing these models, a combination of three elements was used: Petri nets as a means of description, an integrated method and the tool Design/CPN. It was demonstrated that during the phases of system development, covering the system specification through to the final system design, a model based on Petri nets can be used.

REFERENCES

[1] UNISIG: ETCS Subset 026 - SRS. System Requirements Specification.

[2] UNISIG: TIU FFFIS. 97E117 Version 1.0.

[3] Meyer zu Hörste, M.: Methodische Analyse und generische Modellierung von Eisenbahnleit- und -sicherungssystemen. Fortschritt-Berichte VDI. Series 12, No. 571, Düsseldorf, 2004 (In German).

[4] GMA Unterausschuss 1.8.1 Standardisierte Beschreibungsmittel in der Automatisierungstechnik: Glossar. Braunschweig, 1998. www.ifra.ing.tu-bs.de/gma181/glossar.htm.

[5] K. Jensen: Coloured Petri Nets, Volume 1, Monographs in Theoretical Computer Science. Springer-Verlag, Berlin u.a. , 1992.

[6] Design/CPN: Occurrence Graph Analyser-Manual. Version 3.0, Aarhus, 1996.

[7] H.-M. Schulz: The complexity of technical testing. FORMS'98, Braunschweig, 1998.

[8] M. E. Fagan: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, July 1976, pp 182–211, reprinted 1999, pp 258–287.

[9] E. Yourdon: Structured Walkthroughs. Prentice-Hall, Englewood Cliffs, NJ, 1979.

$\sqrt{17}$
$\Omega \int \delta e^{i\pi} = -1$
{2.7182818284}ϑφερτυθιοπσδφγηξκλ
$\chi^2 \sum \gg \approx \lambda$
!